CALS TEST NETWORK

CTN Test Report
91-040

A Path to Tri-Service Use of SGML

May 31, 1991

DTIC QUALITY INSPECTED 4

Prepared for
Air Force Logistics Command
Air Force CALS Test Bed (LMSC/SBC)
Wright-Patterson AFB, OH 45433-5000

19960826 087

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

CTN Test Report
91-040

A Path to Tri-Service Use of SGML

May 31, 1991

Prepared By
SOFTWARE EXOTERICA CORPORATION/CENTECH for
Air Force CALS Test Bed
Wright-Patterson AFB, OH 45433

CTN Test Report
91-040

A Path to Tri-Service
Use of SGML

May 31, 1991

Prepared By
SOFTWARE EXOTERICA CORPORATION/CENTECH for
Air Force CALS Test Bed
Wright-Patterson AFB, OH 45433

AFTB Contact
Gary Lammers
(513) 257-3085

CTN Contact
Mel Lammers
(513) 257-8882

Prepared for
Air Force CALS Test Bed
Wright-Patterson AFB, OH 45433-5000

A Path to Tri-Service Use of SGML

May 31, 1991

Prepared by Software Exoterica Corporation
for the United States Air Force
under Contract to Century Technologies Inc.

A Path to Tri-Service Use of SGML

1. Introduction

This report presents a "road map" of past and future use of SGML[1] within the CALS initiative. It describes the path from the current state of CALS through to full Tri-Service use of SGML-based technology. A thorough understanding of the historical and theoretical basis of using SGML, as well as an understanding of the current status of SGML within the CALS initiative, is required before the next step in the development of CALS-based SGML usage can be competently planned. This document therefore presents a summary of the relevant history and theory behind SGML and CALS, as well as a discussion of its current status, prior to presenting what has to be done next.

1.1. The Role Of History in the Evolution of the CALS Initiative

The CALS initiative is an ongoing development of new standards for the management of technical documentation, from initial creation through all the stages of production to final printed copy or on-line computer access. In part, CALS is an attempt to predict the future of document management within the Armed Forces and by their contractors, as well as an attempt to profitably prepare for that future.

The primary purpose of establishing standards is to provide a medium for allowing public access to developments within the CALS initiative. Viewed in this light, the ongoing revision and adaptation of these standards is not just an inconvenience forced on the community of producers and users of technical documentation by a rapidly changing technology; rather it is a medium by which the major advances that are being made in the technology can be used to advantage by that community.

This view is in contrast to the traditional view of standards as a formalization of accepted practice in a mature technology. It is forced on the technical publications community by both a rapidly changing technology and a perceived need to take advantage of advances in that technology. It requires the technical publications community to adapt to rapid changes, but returns the cost of doing so by keeping the community at the leading edge of new developments.

The best way of understanding future developments within the CALS initiative is to attempt to extrapolate forward from the current state of CALS in light of past developments. Such an extrapolation is what this report represents. An interesting aspect of this review of the history of text markup languages, SGML and CALS, is that expectations for the future have undergone almost continual change responding to greater understanding of the problems inherent in documentation management. This document

[1] The defining document for SGML is *Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML).* ISO (International Organization for Standardization), 1986.

provides the best possible review of what can be reasonably expected to occur in the CALS environment, and contains recommendations for studies of existing components of CALS that will probably further change these expectations.

1.2. Outline

This report develops a practical path towards Tri-Service use of text markup languages. It consists of:

1. a discussion of the historical and theoretical basis of the use of text markup languages, together with a description and discussion of the function of SGML, and role of non-SGML components in the design, use and processing of text markup languages,
2. an overview of the current state of markup languages within the CALS initiative,
3. a presentation of the steps necessary to overcome the problems currently encountered within the CALS initiative, and
4. a presentation of how the proposals in this document should be put into practice.

1.3. Credit

This report was prepared for the United States Air Force CALS Test Network by Sam Wilmott and other members of Software Exoterica Corporation under contract to Century Technologies Corporation. The report represents the current state of research at Exoterica, and parts of this report have been adapted from Exoterica's training materials and publications.

2. A Basis for a New Generation of Markup Languages[2]

A new generation of Tri-Service text markup languages will have to solve the major problems presented by the present generation of MIL-M-28001-based markup languages. The main thrust of this report is that the next stage in the development of SGML usage in the CALS environment must be based on a new, unified approach to markup language design.

Before a unified approach to markup language design can be developed, the nature of markup languages and the way in which they can be used must be thoroughly understood. Each stage in the development of text encoding, of SGML and of the CALS initiative has brought to light new understanding of what markup languages are and what can be done with them. The proliferation of variant CALS markup languages, some strongly, and some more loosely based on those defined in MIL-M-28001, is partly due to the incompleteness of MIL-M-28001, and partly due to the increasing awareness of new capabilities and new requirements.

The following sections describe and discuss the history and development of markup language use, which forms the basis for the use of SGML in the CALS environment. The discussion is phrased in terms independent of the specific kinds of markup languages required for Tri-Service use, so as to allow a fuller discussion later on of what directions current CALS technology can take.

2.1. The Origins of Markup Languages

Text markup languages predate SGML and CALS. They even predate computers.

2.1.1. Copy Markup

Text markup languages were used by the typesetting community long before the introduction of computers. Copy editors pencil symbols and abbreviations on handwritten or typewritten documents. There are typically two types of marks: those that indicate corrections to the handwritten or typewritten document, and those that indicate to the personnel actually doing the typesetting the appearance of each part of the printed document. The first type of mark is that most often seen by writers, when their draft copies are returned from proofreading. The second type of mark is more the business of a print shop. When a marked-up document is sent for printing, it is often accompanied by a "style sheet", which lists both features of the typeset document as a whole and common features of regularly occurring elements of the document, such as the font size used for figure titles. Sets of marks used for these purposes have been standardized, so that most writers, editors and

[2] The text of this chapter has been adapted, with permission, from training material of Software Exoterica Corp.

typesetters know that "#" stands for "add space", "stet" means leave alone, and a caret means "insert here".[3]

The marks used for copy editing constitute a text markup language. The major characteristics that it shares with other markup languages (including CALS markup languages) are the following:

1. Marked-up documents consist of text interspersed with marks.
2. Marks are "punctuation": they either surround text, are placed before or after text, or between pieces of text. Marks can also surround or be adjacent to other marks, in a similar manner to punctuation, in the same way that quotes can surround a period that ends a sentence.
3. Marks are usually very short, a single letter or symbol, although they can be longer. Special symbols are used for some marks ("#"), although many consist of words or abbreviations ("stet", "uc", "cap") or a combination of the two ("sp" in a circle for "spell out").
4. The set of marks is defined in two parts: in *A Manual of Style*, for example, there is a table containing the marks themselves as they would be handwritten by a copy editor or stylist, and the accompanying text explains what each mark means in terms of the effect it has on the text it surrounds or is embedded in.
5. The markup language is at least partially standardized within the community in which it is used.
6. Different marks deal with different characteristics of the text. A copy markup language is entirely concerned with the format and correctness of the text when presented to the reader, and has separate marks for format and marks for correcting text. Other characteristics of the text, even those regarding such obvious book production issues as what words and phrases should be placed in an index, do not have defined marks associated with them. This does not stop authors from adding to the markup language, by as simple a means for example, as circling words and phrases to be placed in the index.
7. The marks are independent of the text of the document. From the point of view of copy markup, words are words. The typesetter need not have any understanding of the text being set; it may even be in a language unknown to the typesetter. The typesetter need only understand what is conveyed by the marks.
8. On the other hand, the purpose of typesetting a book is to make it more readily available to readers, who are the ultimate users of the marked-up documents. These readers are primarily concerned with the text of the

[3] For a good example of standardized copy marks, see *A Manual of Style*. pp. 71-76. 12th Edition. Chicago: The University of Chicago Press, 1969.

book. The formatting that resulted from the marks, the spacing, font changes, boldfacing and italicizing, serve to make the book easier to read.

2.1.2. Punctuation as Markup

Common punctuation and spacing in printed text serves as text markup language of sorts: spaces separate and delimit words; periods terminate sentences; dashes or commas surround elliptical phrases. Punctuation shares the key characteristic of other markup languages, i.e. punctuation can be interpreted without understanding the text of the sentences. For example, the punctuation of French sentences can easily be understood by English-only readers, whereas Japanese, in which there often are no word breaks and which uses a different system of punctuation, looks like vertical rows of marks, with no apparent structure or meaning.

The "punctuation" characteristic of markup languages is what distinguishes them from human language and from computer languages. Very close analogies can be drawn between otherwise radically different markup languages because of this common characteristic. Punctuation carries relatively little of the information in printed text. Printed text without any kind of punctuation at all is very hard, but not impossible to read, and so long as it is in any way readable, it can convey most, if not all of the information intended. Although the computerized use of markup languages is based in part on increasing the amount of information in the punctuation, so as to make the information more readily available to computer applications, the limitations of markup languages have to be appreciated to realize what can be done with them.

An important characteristic of any type of punctuation is that it must surround, precede, follow or be embedded in what it is marking. The placement of punctuation depends on the nature of what it is marking. For example:

1. Quotation marks surround quoted text. This is because quoted text can consist of a word or phrase embedded in a sentence, with no other way of distinguishing its boundaries. As well, quoted text can appear inside quoted text, requiring distinct marks for the start and the end of each piece of quoted text. The use of single (' and ') and double (" and ") quotes is used as an extra visual clue for the human reader, although use of a single style would be unambiguous.
2. Commas or dashes surround elliptical phrases. Elliptical phrases are typically not embedded within each other, and so the same mark is adequate to start the elliptical phrase and to end it. When elliptical phrases are embedded, the reader usually uses the meaning of the text to distinguish between the possible cases. When an elliptical phrase is at the start or the end of a sentence, the leading or following comma or dash is omitted. A period never immediately follows a comma, for example.

3. Sentences are terminated by a period, question mark or exclamation mark. The start of a sentence is unmarked other than by capitalizing the first word. Sentences need only one piece of punctuation because they cannot be embedded within other sentences. In the case of a sentence, an end mark is adequate. The start of a sentence is signalled by the start of a paragraph or by the fact that text immediately follows the end of another sentence.

4. Commas separate alternatives when they are listed in a sentence. Leading and trailing marks are not needed if all the alternatives are grammatically appropriate at that point in the sentence. For example, this sentence lists the sequence of words apples, oranges and pears twice, without leading or trailing punctuation on the list "apples, oranges and pears" the first time, but including such punctuation (the quotation marks) the second time.

5. Space is used as punctuation. Vertical space is used to separate paragraphs. Indentation is used to mark the start of a paragraph. (In the latter case, indentation is often adequate without any other visual clues. The spatial mark in this case functions similarly to the period ending sentences, but appears at the start of what is being marked rather than the end.)

Choosing an appropriate placement of marks is as important as the form of the marks when designing a markup language, be it punctuation for a human language or tags in a MIL-M-28001 variant markup language. There are trade-offs that have to be made:

1. Unnecessary markup and marks that contain an unnecessarily large number of keystrokes can make a marked-up document hard to read. If the text being marked up is smaller than the marks, what has been marked up can be hard to find.

2. Unnecessary markup and marks that contain an unnecessarily large number of keystrokes can also substantially increase the amount of time it takes to enter or modify a document.

3. Too great a dependence on symbols for marks, as opposed to descriptive words and abbreviations, can overload the capacity of the human reader's short-term memory and greatly slow down and reduce the accuracy of the reading process.

4. Unexpected dependencies between marks can make a marked-up document hard to interpret. For example, commas mean different things according to where they are used in a sentence, and the same symbol can be used unambiguously to close a quote and to indicate inches. On the other hand, overuse of these marks could cause them to be misinterpreted, such as inches inside quoted text. This is the cost paid for

the advantage of avoiding unnecessary markup and unnecessarily large marks.

5. Dependencies between marks increases the time and complexity of processing text. This should not be a consideration when designing markup languages for human use, but can be a consideration when designing languages for interchanging data between computer programs.

2.1.3. Early Text Formatting Languages — Procedural Markup

Text formatting languages in the 1960's were based, in part, on copy markup. Individual codes were placed in the stream of text wherever a change of format occurred. Text formatting software would then run over the text and the interspersed codes, and change its formatting action on text based on each code encountered. As a markup language, an early text formatting language had the following characteristics:

1. Codes were "low-level", in the sense that each code dealt with one aspect of the formatting. For example, bold-face could be turned on or off. To set a heading, on the other hand, each of the font, point size, horizontal alignment and spacing characteristics of the text had to be set separately. Once the text was set, these characteristics had to be reset for the paragraphs following the heading.

   Another aspect of the "low-level" of these languages was the requirement to specify explicit measurements. For example, when extra vertical space was required, the exact amount of space had to be specified.

2. There was little or no inheritance of characteristics. For example, to get bold-italic text, a bold-italic code was required; turning on bold, and then turning on italic, would, in many systems, leave the software setting text in italic, but not boldface.

3. The meaning of codes was related to operations on text, rather than to the characteristics of text. If a title was to be centered in a line, the "center this line" code was used. There was usually no "this is a centered line" code. This distinction is sometimes unimportant, but usually its effect is apparent when using such a language. In the early text formatting languages, for example, a separate "set this line ragged right" code had to be inserted at the end of each paragraph for which all the other lines were flush to both margins.

These characteristics are those of a *procedural* markup language. A procedural markup language is one in which the marks are instructions for processing the text. Such languages are relatively easy to implement on computers, but are generally inconvenient for humans to use, due to the amount of detail that has to be thought about when using them. This necessary concentration on detail also encourages errors on the part of the human operator, for even with the greatest care, it is often difficult to predict the effect of many text formatting instructions.

Procedural markup languages were primarily used for describing text formatting. Other shortcomings of procedural markup languages, such as their inflexibility, became clearer later on when other uses were made of them.

2.1.4. The Evolution of Text Markup — Descriptive Markup

The clumsiness of using these early languages was soon partially alleviated by adding a *macro processor* to the text formatting system; this allowed commonly occurring groups of codes to be packaged together. A macro processor allows human users to define their own marks, called macros, which when found in text are replaced by a string of text and (possibly) other marks. The replacement constitutes the definition of the user's macro.

Often macros can have arguments. A macro that has arguments actually consists of a set of marks, and the text surrounded by these marks constitutes the macro's arguments. The arguments can then be examined and inserted into the replacement of the macro. Macro processors allow user definition of complex marks. They also allow for the specification of commonly occurring quantities in one place in a document, such as vertical spacing amounts: macros can be used as the symbolic name of what they replace.

Macro processors were a popular subject of study in the mid-1960's in the computer-related departments of academic institutions, and it was in such institutions, and in companies that produced technical documents and which were in touch with such institutions, that macro processors were primarily used. Commercial text formatting systems tended to make less use of macro processors, and even now stand-alone computer-based typesetting systems, with all the shortcomings of early text formatting languages, are in widespread use.

Once macro processors started to be used, it was quickly discovered that they could be used to define a whole new kind of text formatting language: a *descriptive* markup language. A descriptive markup language is one in which the marks are associated with text and other marks and in which the marks specify some characteristic of the associated text or marks. Formatting or other operations are not specified by the marks, but are a consequence of the characteristics of text. For example, a paragraph would simply be marked with a "paragraph" mark, usually at the start.

Descriptive markup has the immediate benefit of reducing the amount of detail that has to be considered. Formatting considerations can often be ignored by the person entering or editing text: a paragraph is just a paragraph, it does not matter how it looks. Punctuation can be considered a form of descriptive markup: it marks a sentence as a sentence, but does not indicate in any way what is to be done with a sentence.

Although descriptive markup increases the convenience of entering marked-up text, it does not solve all the problems of the early text formatting systems. In particular:

- Because the processing software was only aware of the procedural markup, it could not take advantage of the information provided by the descriptive markup. There really is more to a chapter title, for example, than just that it is in boldface type and centered. There is the fact that the text following it expands on the topic described by the text of the title.

  A more immediate consequence of the processor's ignorance of the descriptive markup is that it was usually necessary to use procedural markup when a particular macro's replacement produced inappropriate formatting. For example, an explicit page break is often required to break text at appropriate spots. This is very inconvenient when text is undergoing continual revision, and the page breaks have to be moved for each revision. The inconvenience is compounded by the common necessity to format and print the document one or more times before appropriate page breaks can be determined.

- The descriptive markup is often directed at one form of processing: text formatting. The marks in the marked-up text are of no use in any other context. The effect of this is illustrated by the fact that users of marked-up documents still often request "stripped" documents, with the marks removed, because the marks actually impede human or machine use of the document. This request is often made when the other context is text formatting in a different style or when a different computer system is used. Thus descriptive markup often retains the inflexibility of procedural markup.

Later text formatting systems made direct use of some of the descriptive markup. With the introduction of "desktop publishing" and "style sheets", a more descriptive markup-oriented approach became more common. Style sheets in word processors and desk top publishing systems allow a user to describe the formatting for each paragraph type. They are similar to the style sheets traditionally used in association with copy markup, but differ in assuming less expertise from the interpreter of the style sheet, in requiring more specific information to be provided, and in being less flexible in what can be requested.

2.1.5. Preparing Input For Data Base Systems

Until recently, data base systems have been primarily concerned with the storage, representation and processing of numerical information. In these systems text has a limited function, usually as labels for rows or columns of numbers. It is only in the last decade that significant attempts have been made to apply data base technology to textual documents. Prior to that, text was stored in file systems as "flat" text files, with nothing known about them

other than what could be implied from the fact that the different parts of a document were often stored as a hierarchically structured set of files. Text files stored this way were most commonly only used for producing printed pages or page-like displays on computer screens.

The most successful and widespread use of text files for other than this purpose was the full inverted text file. This was a way of storing text placing all the words in the text (except possibly some common words that only perform a grammatical function, called "stop words", such as "the", "an", "and") in an index, with each word in the index pointing to all its uses. This form of storage results in very efficient access to any word or combination of words in any part of a large collection of documents. It also lends itself to "Boolean" enquiries, based on combinations of tests, that are common in data base enquiry systems.

Data base systems used input systems based on "forms", fields on a computer screen into which users typed text. A set of data was organized in "records", packets of information, with only a single hierarchical level of structure within records, called "fields". Record-oriented and fielded data could be transmitted from computer to computer by simply tagging each record and field with a code indicating its type, and its length. The use of fixed length records and fields, fixed sets of field types, and only one record type meant that field and record types and lengths could often be dispensed with. As well, each application using a data base system was usually responsible for its own file import and export, so there was good reason to go with the simplest method in each case. The rigid format used for data transmission meant that there was generally no need for a markup language, and that when there was, it could be very simple.

Data base systems and text formatting systems are fundamentally different in that the former provides a basis for storing and retrieving data, and the latter a basis for processing it. Data base systems typically provide simple means for entering and displaying data, but depend on user-written computer software to provide any significant processing of the data. In a classical data base system the record and field structure provides very little information that affects processing compared to what is provided by the contents of the fields. The contents of the fields, text or numbers, are acted on by the user's applications, producing the wide variety of computer software based on data base systems. Text formatting systems, in contrast, provide most of the processing of the text, and this processing is based on marks surrounding the textual data. The text itself, typically has little or no effect on processing, other than providing the characters to be typeset.

The widespread use of "spread sheets" on microcomputers introduced the need for a more flexible way of transmitting data. Spread sheets were no more sophisticated in the kind of, and information about, data they could store than were data base systems. On the other hand, at the expense of imposing a rectangular view onto data, they provided simple programming languages

that made it easy for users to specify processing of the data in the fields. Any field could be a number (displayed in any one of a large number of formats) or text, or could be the result of a computation performed on other fields. The result, for data interchange, was a simple markup language, used solely for the representation of spread sheet information.[4] A lot of the information the marks encoded was formatting information, and the organization of the interchange files as a sequence of commands and text was very much in the spirit of early text formatting languages.

2.1.6. Text Interchange Languages

The interchange of spread sheet information was one of the first widespread uses of data interchange between different pieces of computer software. The primary motivation for this was the requirement on early small systems for a clear-text way of transmitting spread sheets between systems. A *clear-text encoding* is one in which only standard graphic (printable) characters are used, and in which the marks are easily recognizable (usually human-readable). The clear-text encoding meant the primitive, early microcomputer file systems could support it, and that application-specific import and export programs were not required. As well, a clear-text format meant that information could be easily read by other programs, such as ones which could print graphical representations of spread sheet information.

The major factor in the introduction of common interchange formats was heavy competition in the microcomputer software market. Each new spread sheet program had to be able to read the files produced by all the other spread sheet programs so that users of those other programs could move to using the new program. Because every program could read every other program's files, an environment was created in which the formats used by the most commonly used programs could become *de facto* standards.

Word processing systems on microcomputers developed a little later than spread sheets. Each of the early word processors and desktop publishing systems used a proprietary format for documents. This was usually a *binary encoding*: one in which numeric values of characters represented some code based on the number rather than the characters themselves. The same motivations came into play as in the spread sheet arena. In the word processing and desk top arena, the drive to a clear-text format was not based on the limitations of the file systems but on the need to import and export information from other software. The need to write a multitude of file conversion programs made simplifying the task a major requirement. Currently, most such systems have their own clear-text format, and some of

[4] A description of the original spreadsheet interchange format and a few of its variants can be found in *File Formats for Popular PC Software*, by Jeffrey Walden. New York: John Wiley and Sons, Inc., 1986. Pages 108-111 contain a description of the original format.

the more popular ones are becoming used widely enough to be considered standards.[5]

2.2. Capturing Information

What characterizes all the methods of capturing information described in previous sections of this report is that each captures information for a specific purpose. This purpose is usually formatting text for a printed page or for a page-like image on a computer screen. Spread sheet interchange languages capture information for simple formatting and for linking information together to allow computations to be done. The marks in a markup language contain information, either in the marks themselves or by identifying the text that they surround or to which they are adjacent. The text between the marks is also information, although in most text-formatting-oriented markup languages, the characters are themselves the only information in the text. That is, the information carried by a letter "a" is that a letter "a" is to be printed. The fact that it is part of a word, a sentence or a report is of no interest to the markup language.

The question "What is information?", when asked about a marked-up document, can only be answered by looking to the origin of the question. When asked in the context of a text formatting system, the information is formatting information: paragraph boundaries, required spacing, font, type size, etc. When asked in the context of a data base system, the information is that about which enquiries are expected to be made. The answer to the question is much clearer for text formatting systems than for data base systems because the kind of processing done by formatters is better known and the role of the data in that processing is well known.

What information there is in a text document depends on what is going to be done with the text. In addition to text formatting and data base retrieval, there is another common use of the data: people read it. Text formatting information allows computer software to present text in a readable form. This means that text formatting information is always an important part of what needs to be captured by a markup language. Data base information allows computer software to perform other tasks using the text. So the decision to capture data base information using the markup language depends on what uses other than straight-forward reading are going to be made of the text once processed.

In designing a markup language to capture information, care has to be taken to only capture the information that can be usefully processed. Even when a lot of markup is done, most of the information in a text document is contained in the words of the document. In this regard, text is different from

[5] A case in point is Microsoft's RTF (Rich Text Format), described in *Microsoft Word For Windows and OS/2 Technical Reference*, by Microsoft Press. pp. 381-409. Redmond: Microsoft Press, 1990.

the rows and columns of numbers that are usually stored in a data base system, where the number 1 (one), for example, is practically meaningless without some information about what it is one of. Most of the text can be understood without any indication of what it means provided by the markup language. There is, in fact, so much information in text that on one hand only a small part of it can be understood in any non-trivial sense by computer software without markup, and on the other, it is impractical to mark up more than a small part of it.

2.3. What Is a Language?

Before discussing SGML and its application in the CALS environment, this section will present a more general discussion of languages, and place markup languages in wider context. This will allow the reader to more readily understand the potential of SGML-defined markup languages, especially as they can be exploited for Tri-Service use.

2.3.1. Grammar

Human languages and computer languages share the need for an underlying grammar. A language's grammar is what determines the interrelationship of individual pieces of information. Textual information is generally built out of small pieces of information. The grammar determines how the pieces are put together. A grammar also allows the individual pieces to be recognized in a stream of text. This process of recognizing the pieces and their interrelationships is known as *parsing*.

A grammar for the English language allows one to *parse* a sentence and recognize the nouns, verbs and other words and phrases. The grammar also determines which nouns form part of the subject, and which the object. There are two ways in which a word's identity is determined. First, the word itself determines, or at least suggests, what kind of a grammatical item it is: "foot" is usually a noun, for example, although it can be a verb. A grammatical item is an identifiable part of something being parsed. For a grammar of English, the term *part of speech* is usually used. Second, the grammar is used to find the relationship between the word and the words and punctuation around it: in "I can foot the bill.", for example, "foot" is a verb, because it follows "can" and because the sentence already has a subject ("I") and an object ("the bill"). Because, in English, a word can often serve as a noun, adjective or verb, the grammar is especially important in helping recognize the meaning of individual words. The interrelationship between the words and phrases in a sentence is called the sentence's *structure*.

A grammar is usually described by a set of *productions*: rules that describe each grammatical construct in terms of others. For example, in English, one form of a sentence consists of a subject, followed by a verb and an object. A subject can consist of a noun, as can an object. These rules are often written in a form similar to the following:

sentence → subject, verb, object

subject → noun

object → noun

A parsed sentence can be graphically represented by a *parse tree* (by analogy with a family tree, rather than one that grows in the ground), in which relationships are represented by branches, and grammatical items by nodes, with the name of the grammatical item used to label the nodes. Figure 1 is a simple parse tree for the sentence "Children like candy", in which the grammatical item "sentence" is made up of three other grammatical items.

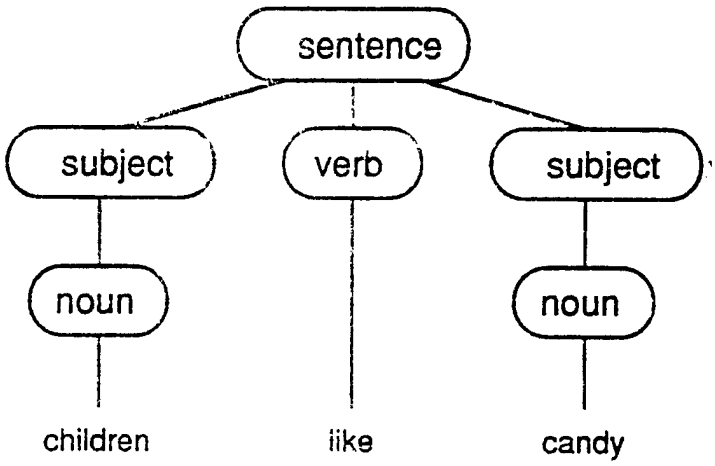

Figure 1 — A Simple Parse Tree for an English Sentence.

A parse tree imposes a hierarchical structure on a sentence. A hierarchical, tree-like structure is chosen primarily because it is a convenient way of organizing information. Non-hierarchical interrelationships, such as the requirement that the subject, verb and object be consistent with each other in their meaning, and that words like "it" and "they" refer to parts of previous sentences, are not always convenient to represent in a hierarchical form. Nonetheless, the convenience of the parse tree usually overrides these shortcomings. More sophisticated ways of representing the results of parsing tend to use parse trees, but with extra information labelling the branches and nodes. For common use, parse trees with accompanying explanatory text, using the labels that appear in the parse trees, are the most convenient graphical representation for the results of parsing.

A parse tree provides a lot of information about the structure of a particular sentence, in a form that is easy to comprehend, but does not say very much about the language being parsed. This is because it describes only one of, in the case of English, many millions of possible sentences. Productions of the sort shown above contain much more information about English:

productions should be used in preference to parse trees except when particular examples are being examined.

Parsing sentences is something that most people learn in school, although many adults have long since forgotten what a participle is. Parsing is something that people usually do without thinking about it: we understand sentences easily enough. As well, parsing is something that is commonly applied to structures other than just sentences. An important example is the layout of a printed page, which has a very distinct structure. We determine which pieces of text are headings, what the headings apply to and which paragraphs are discussing the same subject in a similar manner to parsing a sentence. Headings are centered, in boldface type, or both, and paragraphs are indented, so there is little trouble recognizing them. Nonetheless, these clues are not necessary. In typewritten material, a heading of sufficient length may look just like a short paragraph, but the grammar of a document, that indicates, for example, that a document starts with a heading, and that paragraphs are made up of complete sentences, allows headings and paragraphs to be easily distinguished. The grammar of documents also indicates that headings precede what they head, so associating a heading with the textual explication of the topic listed in the heading is easy, once the headings have been identified.

These characteristics of grammars for human languages also hold for computer languages in general, and markup languages in particular.

2.3.2. Syntax and Semantics

A grammar describes the syntax of a language. The syntax is the physical form of the language. A parse tree, for example, is a graphical representation of the syntax of a sentence. The one thing that the syntax does not do is associate any meaning with a sentence. All sentences with a noun as a subject, noun as object and verb between have the same syntactic structure.

The "meaning" of a language is called its *semantics*. A grammar of the sort described above does not associate any semantics with grammatical items. Semantic association, or definition, is usually done using English text, which describes the semantics of each grammatical item, often with variant definitions depending on the context of the grammatical item. The names of the grammatical items used in the productions can be used in these definitions. Doing so helps to make the definitions more precise. The English text can also contain constraints on the form of the language, when the grammar is not restrictive enough in the combinations it allows, or when there are interdependencies that cannot be described in the kind of grammar being used.

2.3.3. Languages and Meta-Languages

The sample productions for English sentences in an earlier section are themselves written using a simple language: the language used to describe a

grammar. The "grammar language" can itself be described in terms of productions:

grammar → sequence of productions

sequence of productions → production

sequence of productions → sequence of productions, production

production → name, right arrow, sequence of items

sequence of items → item

sequence of items → sequence of items, comma, item

item → name

item → string in quotes

right arrow → "→"

comma → ","

As this grammar illustrates, grammatical items (parts of speech) can have more than one definition, and items can be defined in terms of themselves. For example, a "sequence of productions" is one or more "productions", which is achieved by defining a sequence of productions to be either a production or a sequence of productions (which itself may be either of these) followed by another production. All grammars other than the simplest use these techniques. They are especially powerful when used with computer languages, including markup languages, which lack the rich semantics of human languages.

Grammars can be defined in a number of ways. The previous example is written in a style familiar to school children. Computer programming languages are commonly defined using a notation based on BNF (Backus-Naur Form). Early forms of BNF used "<" and ">" around the names of grammatical items, and "::=" in place of "→". More recent notations tend to look more like the above. The exact notation is not important, the form of the productions (a thing is defined as a thing, thing, etc.) is.

The "grammar language" is a *meta-language*. A meta-language is a language used to define other languages. The language of productions is a meta-language. The distinction between languages and meta-languages is usually quite clear for computer languages. For human languages it is less clear, because most commonly, the type of language used to describe a human language is human language, usually the very language being described. Human language is also used as a meta-language in defining many computer languages. For example, if English is used to define the semantics of a computer programming language, then English is being used as a meta-language.

2.3.4. Surface Structure and Deep Structure[6]

Any sentence or document is only a representation of the information it contains. Language is a medium for carrying information. The syntactic structure of a sentence or marked-up document is a consequence of the language in which it is encoded. This structure may or may not have anything to do with the structure of what the sentence or document is about. For example, the structure of English sentences when one is speaking is the structure of English sentences, not the structure of what one is talking about. The three part structure of the sentence "Children like candy" has nothing to do with children or candy. In other words, structure is syntax, not semantics. The semantic information conveyed by structure is determined by interpreting the syntactic structure in terms of the semantics of the language in which the information is encoded.

In computer encoding of data, it is often the case that the syntactic structure of data is equivalent or close to that of the information being coded. Information inside a computer is often stored in a hierarchical format for the same reasons of convenience that parsing is usually represented in a tree-like manner. In this case it is easy to design a markup language that is close in structure to the structure of the information.

It is much harder to formulate a syntactic structure for a markup language that is close to the form in which the data will be used when marked-up data is expected to be used in more than one context. Even two different text formatting systems may need to have data presented to them in two substantially different forms. When a document is both to be published as a book, and loaded onto a data base system, it is very likely that the information will need to be represented in two different ways, and if a common source document is to be used for both purposes, it is likely that some form of translation is required from the source forms and the different forms used for processing the document. As well, when information is processed, there is a possibility that not all of the markup and text will be of interest to the computer system doing the processing, or that information used in two or more ways will need to be duplicated, with different markup required for each copy. For example, a data base system may not be interested in all the formatting information. Similarly, a text formatting system may not be interested in the information used when data base enquiries are made of the data, and may require separate copies of titles placed in running headings. As a consequence, it is often necessary to remove some text from a document that is to be processed, and to duplicate other text.

---

[6] The linguistic basis for the understanding of the kinds of structures captured by text markup languages was developed by Noam Chomsky. A key work describing his theories is *Aspects of the Theory of Syntax*, by Noam Chomsky. Cambridge: The M.I.T. Press, 1965.

Each markup language provides a different representation for the same information. This is analogous to the fact that different human languages have different grammars. Parse trees for equivalent sentences in different languages are usually different, even for closely related human languages such as English and Dutch. Differences are managed by translating from one language to another. The only limits on translation are the requirement for a competent translator, and the fact that the result of translation can, at best, only contain as much information as what is being translated.

Humans and computers deal with particular representations of languages. The structure of these languages is called *surface structure.* Surface structures are used to make data transmission (communication) easier. Some examples of language features that make communication easier are:

1. Linear encoding: A linear encoding is one in which the smallest pieces of a message are strung together, one after the other. "Flat" text files on computer systems are a linear encoding. Text is entered at a keyboard as a linear encoding of keystrokes. Human speech is a linear encoding of sounds.
2. Regular punctuation: "Regular" punctuation is punctuation that marks the end of something and the start of something new, like the periods that end sentences. An example of non-regular punctuation is the quotation mark, which is used to surround the structure it "marks up", but does not end what surrounds the quotation, and which can be used in a nested fashion (quotes within quotes). Although hierarchical structures are a powerful tool for thinking about the structure of information, heavily nested structures are difficult to type in without a lot of visual clues as to what is inside what.
3. Space: Horizontal and vertical space in printed documents, and different rates of speech and pauses in speech, aid comprehension, even though punctuation may be adequate for a computer.

In contrast to surface structure, *deep structure* is the structure of information itself. It is the set of interrelationships between the parts of information, in a sentence, or in a marked-up text document. The deep structure is what must be preserved by translation, and is what is lost when a translation is incomplete.

When looking at the design of a particular text markup language, there are two ways in which it can be evaluated:

- A markup language can be examined for the convenience and accuracy with which marked-up documents can be entered, edited and read (for example, for proofreading purposes). This is an examination of the markup language's surface structure.

- A markup language can be examined for what information it is capturing. This is an examination of the markup language's deep structure.

2.3.5. Classes of Languages

Computer processing of languages divides languages into classes, which are distinguished by the ease with which they can be parsed (by computer), and the richness of interrelationships between grammatical items. The four major classes of languages, in increasing order or power and complexity are:

1. A *regular language* is one in which things can come one after the other, but in which there is no nesting.
2. A *context-free language* is one in which nesting is allowed, but in which a grammatical item has the same definition no matter where it is used in the grammar. For example, a context-free language could allow nested quotations.
3. A *context-sensitive language* is like a context-free language, but which allows a grammatical item's definition to depend its context. For example, a context-sensitive language could allow the definition of "noun" to be different depending whether it was part of a "subject" or part of an "object".
4. A *phrase structure language* is any language that can be processed by a computer.

Each class of language includes all weaker classes. For example, the set of all context-free languages consist of all regular languages plus some others that are not regular. The set of productions used to define most computer languages constitutes a context-free grammar. The additional human-language explanations place the languages in a context-sensitive or phrase structure class.

A further classification of languages is used for context-free languages. One of these classifications is *LL(1)* (pronounced "el el one"). An LL(1) language is characterized by each grammatical item being recognizable by the first grammatical item within it (LL(1) means "parsing by examining from Left to right, looking at the Leftmost one (1) character"). For example, placing a mark at the start of every paragraph is the sort of thing that would be done in an LL(1) language: the initial mark is what the paragraph is recognized by. On the other hand, a language that distinguished normal sentences, ending in a period, and questions, ending in a question mark, as distinct grammatical items would not be LL(1) (it would be LR(1): Left-to-right, looking at the Rightmost one (1) character).

LL(1) grammars are popular because they are simple to use and yet allow powerful languages to be defined. They have the added advantage that a *parser* for an LL(1) language is easy to implement on a computer. Markup

languages tend to be LL(1) even more often than computer programming languages, because of their "punctuation" nature.

LL(1) grammars are especially interesting in the CALS environment because markup languages defined using SGML are LL(1). An SGML DTD (Document Type Definition) is an LL(1) grammar. SGML element declarations are "productions", in a slightly different notation than that normally used for defining computer programming languages. For example, the following two productions are essentially saying the same thing, even though one is written in an early form of BNF and one in SGML:

<sentence> ::= <subject> <verb> <object>

<!ELEMENT sentence - - (subject, verb, object)>

SGML can be used to define a wide variety of LL(1) text markup languages. The few limitations SGML places on which LL(1) languages can be defined are usually consistent with good design principles: most of the limitations are designed to discourage the design of languages which seem to be ambiguous to their users, even though, from a strictly technical point of view, they are not.

2.3.6. Marks

A lesson that originated in the field of psychology rather than in computer programming, but which impacts the design of both computer programming languages and text markup languages, is that there is a limit to how many different pieces of information that can be held in human short-term memory. This number has been found to be about seven.[7] This is why telephone numbers have seven digits, and seven distinct tones in an octave (the eighth repeats the first but is an octave higher). This is also why users of icon-based computer software interfaces start to use words and names in place of individual icons when there are more than a small number of items on the computer screen at one time.

A consequence of this observation is that about seven different types of punctuation can be used before some significant amount of memorization is required (i.e. to place the information in long-term memory). Human beings are much more efficient at remembering language than they are random marks. So beyond a certain point, meaningful words should be used as part of markup.

This limit on short-term memory is why most computer programs read like very poorly written English, with only a small vocabulary of special marks, and then only for very common constructs. Likewise, in a text markup

[7] An important article to read for anyone interested in this subject is "The Magical Number Seven, Plus-or-minus Two: Some Limits on Our Capacity for Processing Information", by George A. Miller, *Psychological Review*, 63, No. 2 (March 1956), pp. 81-97.

language, a few common constructs can be marked-up using punctuation-like symbols, but where there are more than a few distinct possibilities, "tags", incorporating names for grammatical items, should be used.

2.4. Why Use a Text Markup Language?

As has been demonstrated already, there were many text markup languages in widespread use prior to the development of SGML or the initiation of the CALS initiative. Any proposal to develop a new markup language has to justify itself by it providing some advantage over using an existing markup language. In addition, the use of a text markup language has to be justified in preference to some other technique for representing text.

2.4.1. The Advantages of Text Markup Languages

There are a variety of alternatives to text markup languages when dealing with textual documents. Word processors, desktop publishing systems and data base systems each have their own format for saving text and the formatting and structuring information that goes along with the text. Most commonly used text management systems can communicate with other such systems, either by being able to read files stored in the format used by the other systems (usually a binary encoding), with the help of a simple conversion utility that converts one system's format to another's, or with the help of an existing agreed-upon interchange format (usually a clear-text encoding). Thus, for many uses, file formats, tools for entering text in these formats, and tools for displaying these formats in human-readable forms already exist.

Publishing and word-processing-oriented formats are usually limited in their ability to represent information in that they can only associate print formatting structure with text, and only be used in association with a limited range of computer software. Data base file formats often allow for a much more flexible interpretation than only as printed documents, but are usually even more limited in the range of computer software that can be used with them. New ways of storing text are required when an application has requirements beyond these limitations.

A text markup language has a number of advantages when a new way of representing information is required. Primary amongst these advantages is that a text markup language provides a clear-text encoding, so that files can be created and viewed using almost any text editor or word processor on any computer system. No new computer software is required for these functions, although there are limited application domains in which software designed to aid in the entry and editing of specific types of markup languages can facilitate the process.

A clear-text encoding has the further advantage that it makes the job of writing a parser, that produces the same results on a wide variety of computer systems, easier than if a binary encoding is used. If no special advantage is

taken of a particular character set or of the control functions supported by individual systems, then the documents so encoded are potentially highly transportable from system to system. The limit on transportability is whether or not there is, in fact, software available to process documents marked up in a given text markup language.

The availability of software that can read a markup language can be increased substantially by the ready availability of a parser or a parser generator for the kind of markup language being used. A *parser generator* is a piece of computer software that takes a grammar written using a set of productions and produces another piece of computer software which can parse text marked up using the language described by the grammar. The produced parser can be source code for a computer programming language that needs to be compiled, or it can be a set of tables or a sequence of codes, the interpretation of which, by suitable software, results in appropriate parsing actions. The produced parser can be used by other computer software to capture the text and structure of input documents. The other computer software can then process the text and structure in some fashion: format it, load it into a data base system, etc.

An SGML parser is a parser generator for text markup languages. It is not a parser for any specific text markup language. An SGML parser can be used to implement much more complex text markup languages than would normally be implemented without the aid of a parser generator.

In contrast to SGML is ASN.1.[8] ASN.1 is one of the few examples of binary encodings for text that have been standardized. It is a language for encoding the structure and text of a document that uses numbers for the names of grammatical items. The ASN.1 standard provides a way of defining subsets of the ASN.1 grammar that allows symbolic names to be attached to grammatical items, and than allows constraints to be put on the structure of grammatical items. ASN.1 has the advantage that it is easy to write a parser if these subset definitions are not taken into consideration. ASN.1's disadvantages are:

- Special software is needed to produce ASN.1 files. Commonly-available text editors cannot be used.
- Very little structural information is encoded in the ASN.1 grammar. Even when a subset grammar is defined, the types of grammatical interrelationships are considerably less rich than those that can be expressed in, say, SGML.

[8] ISO 8824: *Information processing systems — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1)*. ISO: International Organization for Standardization, 1987 and ISO 8825: *Information processing systems — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*. ISO: International Organization for Standardization, 1987

2.4.2. The Role of Text Markup Languages

Text markup languages can be used in a number of roles:

1. They can be used to encode textual documents.
2. Marked-up documents can be used for document interchange, between different computer software and between humans. In a clear-text encoding, they can be readily transmitted, if necessary by being printed on paper and being read, either by a human or by means of an optical scanner.
3. They can provide a standard format for document delivery. It is practical for a receiver of documents to specify that a text markup language be used in documents received, because it is reasonable to assume that all producers can provide a clear-text encoding.
4. Marked-up documents can be readily archived. Virtually any archiving system can deal with clear-text encoded documents, and archived documents are not subject to the encoding idiosyncrasies of a particular software product.

Markup languages need not encode only textual information. Any information for which a clear-text encoding can be devised can be incorporated into marked-up text documents. If a clear-text encoding can be devised for non-text information, it is often practical to use a markup language to encode it, and for that markup language to be part of a larger language which is also used to markup text.

2.4.3. Alternatives to Text Markup Languages

Until about 1980, most computerized typesetting and print formatting was done using system-specific text markup languages. With the explosion of "desktop" and personal computer systems in the 1980's this changed, so that today there is widespread use of WYSIWYG ("What You See Is What You Get") systems. A WYSIWYG system provides immediate visual feedback of formatting information as text is entered. It is a powerful tool for capturing a specific visual representation for text. At the same time, large-scale producers of printed material continued to use markup-language based systems, most of them developed prior to 1980.

The strength of WYSIWYG is also its weakness. Because it captures a specific visual appearance, it is often difficult to discover those distinctions that are meaningful, and those that are a consequence of how a particular piece of text appears on a page. For example, if the first paragraph in a chapter is not indented, but all the other paragraphs are indented, it may be difficult to identify them as the same type of object. Different uses of bold-faced type may be difficult to differentiate. It is often difficult to take a WYSIWYG document and print it in a different format. The differences between WYSIWYG software products means that the result of even a careful conversion of a

document from one word processor to another often produces an unsatisfactory appearance.

Because of their emphasis on visual appearance, WYSIWYG systems produce data that is generally unsuited to data base use. The use of "style sheets" with WYSIWYG systems makes capturing print formatting information easier, although the lack of any firm constraint in this regard in most WYSIWYG systems means that, in practice, it is common for documents to contain mixtures of style sheet and "do it yourself" formatting. Even style sheets, which are defined in terms of appearance, tend to be used in irregular ways. For example, it is common for an inappropriate style to be used for a block of text because, with a little bit of modification, it produced the best appearance.

The new interest in text markup languages has developed largely in response to the inability of WYSIWYG systems to satisfy the requirements of more advanced and flexible text formatting and data base systems. There is presently, however, a large investment in WYSIWYG technology, both in terms of cost and, for certain types of documents, in convenience of use, which much of the publishing community is unwilling to abandon.

2.5. Markup Languages and Programming Languages

A number of lessons for markup language design can be learned from studying the history of computer programming languages.

2.5.1. The Use of Delimiters

The characteristic of text markup languages that makes them syntactically different from computer programming languages is that the grammar of a markup language defines the "punctuation" for a document, and says nothing about what goes between the punctuation (except that it not be confused with the punctuation), whereas a programming language's grammar defines the whole of a computer program written in that language, except for the text that goes in constant strings. This difference is entirely a consequence of the relative proportion of a document (marked up text document or computer program) that consists of text. For a text document this proportion is high, most of the document is usually text rather than markup. For a computer program this proportion is usually very low. The main visual difference between marked-up text documents is a consequence of the fact that the delimiters in a text markup language are used to identify the marks, and in a programming language they are used to identify the text. For example, in SGML, delimiters such as "<!" and ">" are used to identify declarations and other items of markup, whereas in the "C" programming language, delimiters such as "'" are used to identify text.

The characteristic syntactic difference of markup languages and programming languages must be considered in the design of text markup languages when the proportion of markup becomes high. In areas such as the markup of equations or tables, in which markup can easily predominate over the text,

markup languages have to be designed as if they were programming languages to achieve the most efficient and accurate results.

2.5.2. Universal Languages

Another lesson that can be learned from computer programming languages is the experience that has been developed with trying to develop a "universal language". In the programming language field, there have been attempts to develop two types of universal languages:

- In the mid-to-late 1960's there were a number "universal" programming languages developed. PL/1 (Programming Language One) is the best known of these in North America. These languages attempted to provide all the capabilities of existing programming languages, and to eliminate the requirement for using different languages for different tasks.
- In the late 1960's and in the 1970's there were a number of attempts made to develop a "universal" target code. The idea was to allow compilers for various computer programming languages to compile into a single universal target code. For each class of computer, a single translation program from this universal target code into the computer's native machine language could then be written. This would eliminate the need for writing a separate compiler for each language for each class of computer.

Some of the "universal" computer programming languages are still in widespread use, but more specialized programming languages such as "C" predominate. The primary reason for this is that the "universal" languages were not as universal as first thought. Later requirements and developments in software technology quickly made many aspects of the languages obsolete. The reason these languages are still in use at all is that the substantial investment involved in using a new programming language, both in terms of training and converting existing programs and support tools, introduces considerable inertia into the whole process. There have been far fewer "new" programming languages in the public arena in the last decade than in the previous two (there has been a noticeable drop-off since the mid-1970's), and the new languages generally make no attempt to be inclusive: they are designed to solve a certain class of problems or to support a specific programming style.

"Universal" target codes have likewise gone out of style since about the mid-1970's. The reason here has been the much smaller variety of machine instruction sets on the one hand and the development of programming-language-specific target codes. Programming-language-specific target codes are oriented to one computer programming language or to a small class of languages (although most commonly-used languages, like, "C", Ada, Prolog, FORTRAN and COBOL effectively form a class of one member). A compiler is written for the programming language that translates programs in the

language into a specific target code for that programming language. Separate translators are then written for each class of computer. Programming-language-specific target codes are used in preference to a "universal" target code because no universal code has been devised that satisfies the functional needs of all programming languages and at the same time makes the production of efficient machine code simpler than if a language-specific translation were being written.

2.5.3. Standards Development as a Research Activity

The programming language called Algol 68[9] was intended as a "universal" programming language, to be standardized at the international level, and to replace most programming language use at the time (1968). Algol 68 was developed, virtually from scratch, by a small group under the strong leadership of a project editor. A number of notable discoveries were made during its development, and it has had a significant influence on the development of later programming languages. In spite of all this, twenty years later it has very limited use, and most people in the computer field are not even aware of its existence.

The difficulty with Algol 68 was that a programming language that has a wide use cannot be changed very often, or to a substantial extent, without losing a large investment, not only in operating computer software implemented using that language, but in the knowledge of that language by its users. Algol 68 was therefore unable to incorporate the new developments in programming language design that were taking place at the time, many of which were the direct result of discoveries made while developing Algol 68 itself.

Computer programming languages that succeed in establishing a place for themselves in the computer programming community either fill a vacuum, as did the FORTRAN and COBOL languages in the early 1960's, or have to be used and allowed to develop within a small community of users prior to being used by a larger community and standardized, as was the "C" language. Neither of these options is available as a way to establish a standardized text markup language in the CALS environment. There are numerous technologies for managing technical documents currently available and in use. None of them satisfy all the stated needs of the technical document community in terms of on-line data base access and selection for display or publishing, or in terms of managing large sets of documents, but no new technology yet exists that can cost-justify large-scale conversion. At the same time the primary requirement of a new document management technology is that it be readily available to the Armed Forces generally and to other users of military technical documents.

---

[9] "The Revised Report on the Algorithmic Language Algol 68", in *Acta Informatica*, Vol 5. Fasc. 1-3 1975.

The way to avoid the drawbacks of standardizing newly developed technology is to ensure that text markup languages within the CALS environment are flexible and able to adapt to new and varied requirements. This flexibility must be supported not only by the markup languages themselves, but by the tools used to process documents marked up using these languages.

2.5.4. Object-Oriented Programming

The single innovation that rendered obsolete the main-stream computer programming languages of the 1970's was the introduction and immediate popularity of a number of new programming methodologies. Chief amongst these has been object-oriented programming. Object-oriented programming allows computer systems to be organized with divisions between classes of data objects, with the computation and decision-making logic for each class of data objects associated with the objects in that class. This is in contrast with classical programming in which the major division is between data objects on one hand, and computation and decision-making logic on the other.

The move to object-oriented programming has not shown up only in new programming languages, but in "object-oriented" variants of existing languages and in the use of related methodologies such as "modular programming", in which a software system is formally separated into individual modules of closely related routines and data objects.

2.6. SGML

This section describes why SGML is an important component of the CALS initiative.

2.6.1. What is SGML?

SGML is a "meta-language" for defining text markup languages. It is not itself a text markup language but is rather a language in which the context-free syntax of text markup languages can be defined.

An SGML document (properly called an *SGML document entity*) consists of an *SGML Declaration*, a *document prolog* and a *document instance*, in that order. SGML defines the syntax and semantics of the prolog of an SGML document. A document prolog, written using SGML declarations, in turn defines an LL(1) context-free grammar with which the document instance is parsed: in other words, the prolog defines the syntax of the document instance. For an SGML document:

1. The syntax of the document prolog is the form in which declarations appear in the prolog. For example, a simple element declaration looks like:

   <!ELEMENT title - - (#PCDATA)>

   The syntax of SGML is what says that an element declaration consists of:

(a) a markup declaration open ("<!"), followed by
(b) the word "ELEMENT",
(c) a name, consisting of a "name start character", followed by "name characters" ("title" in the example),
(d) two symbols each of which is a "-" or an "O" ("- -" in the example),
(e) a "content model" in parentheses ("(#PCDATA)" in the example), and
(f) a markup declaration close (">").

2. The semantics (i.e. the meaning) of the document prolog is the grammar being defined. The semantic value of the example element declaration is that:
   (a) the name is the name of an element, markup for which is allowed to appear in the document instance,
   (b) when the element with that name is present in the document instance, it must be surrounded by marks that identify that particular element (the presence of the element cannot be implied by the presence of other elements), and
   (c) when the element with that name is present in the document instance, its content can consist only of text characters and cannot contain nested elements.

   An *element* is what in SGML-defined markup languages corresponds to a grammatical item. It is a single recognizable piece of text, and may in turn consist of one or more grammatical items, called *subelements*.

3. The syntax of the document instance is determined by the grammar defined by the prolog, and by additional rules for parsing document instances stated in the SGML Standard. These additional rules deal with issues such as handling white space, and with a means for inserting comments in a document instance without affecting the semantics of the document.

4. The semantics of the document instance is solely determined by an *application*. Neither the SGML Standard, nor the grammar defined by a document prolog contribute in any way to the "meaning" of an SGML document instance.

   An application is a piece of computer software that, in SGML terms, does all the processing of a document instance other than parsing it according to the grammar defined in the prolog. An SGML document instance does not "mean" anything without either an associated application to provide this processing or a definition of the semantics of the instance in some other terms (for example, in a set of definitions for the grammatical items, written in English). In the latter case, the set of definitions can be considered to be the application, as they provide a means for humans to interpret a document.

In most cases the interpretation of a parsed document instance by an application will, as well as providing semantics, impose further syntactic constraints. For example, in most cases it is invalid for an element to be empty of text or subelements (e.g. a paragraph must contain some text). SGML does not provide means for stating this constraint on a markup language in a prolog, so it has either to be implemented in an application, or stated in the set of definitions that serve as the application.

5. The SGML Declaration defines the form of the marks used in the document prolog and the document instance, and so contributes in part to the definition of the grammar of both the prolog and the instance. The SGML Declaration provides a set of sequences of characters used as marks and used to surround marks, a set of characters allowed in text and in the names of grammatical items, and a set of constraints on the marks and their use (an example of a constraint is placing an upper limit on the length of the name used as part of a mark to indicate a grammatical item). The division of function between the SGML Declaration and the prolog is that the SGML Declaration defines the marks used in the markup language (its *lexical structure*) and the prolog defines the interrelationships between these marks (the language's *syntactic structure*).

   There is a division between the SGML Declaration and the prolog for two reasons:

   — If an SGML document entity is transferred to a computer with a different character set, only the SGML Declaration needs to be modified to reflect the new character set. Thus, the separation of SGML Declaration and prolog simplifies the task of interchanging SGML documents.

   — It is very common for different text markup languages to use the same form for their marks. Doing so makes learning a new markup language much easier than if each markup language used a completely new set of marks. The limited number of characters on a keyboard constrains the set of characters available for use, and so encourages such standardization.

     Using a common form for marks does not mean each markup language need use the same marks. Many marks incorporate easy-to-remember names for grammatical items in the language. For example, the mark "<chapter>" is used in many markup languages to indicate the start of a chapter. (Marks of this sort, incorporating a name for a grammatical item, are called, in SGML terminology, *tags*.) As well, as is noted earlier in this report, the number of "iconic" marks must be kept relatively small to make a markup language usable, but for each markup language, the iconic marks can mark different types of grammatical items. (An example of

iconic marks is the use of brackets, "[" and "]", to surround keywords of a certain class. Marks of this sort are called, in SGML terminology, *delimiters*.)

Using a common form for marks is similar to using the same letters, punctuation and conventions for separating words by spaces and line breaks in different human languages, and to using the same form for numbers in different languages. Doing so makes it much easier for individuals to use more than one language, and makes it possible to use the same technology (e.g. typewriters and computers) for different languages with a minimum of difficulty and expense.

The syntax of an SGML Declaration is defined solely by the SGML Standard. Its semantics consists of the set of delimiters and the form of the marks it defines, the constraints it places on those marks, and the set of characters it provides for entering names and text.

The definition of a grammar in a document prolog is called a *document type definition* or *DTD*. A document type definition is contained within a *document type declaration*, which is a declaration that usually starts with a markup declaration open and the keyword "DOCTYPE". A text markup language, its syntax and its semantics, together with the set of document instances that use the language, is called a *document type*. The SGML Standard allows for more than one grammar to be defined in a prolog and for application-specific information to be included in a prolog, but the CALS standard that uses SGML specifies that these features not be used.

2.6.2. Common Misconceptions About SGML

There is widespread misunderstanding of the role of SGML. This report has attempted to provide a conceptual groundwork using which the reader can avoid most of these difficulties. The following is a summary of some of the more common misunderstandings about SGML and its role as a text markup meta-language.

2.6.2.1. Angle-Bracket Languages. The common misconception that an SGML-marked-up document consists of text intermixed with "tags", names of elements surrounded by "<" and ">", is caused by the use of a common set of marks. This is similar to considering two written human languages to be the same if they use the same alphabet. Its syntactic structure and semantics characterize a language, not its marks.

There are many text markup languages that are "angle-bracket" languages (i.e. in which the marks are all surrounded by "<" and ">") but are not SGML-defined markup languages. Many text formatting languages use this

convention.[10] For any given angle-bracket language, it may or may not be possible to write an SGML Declaration and prolog that describe a language which recognizes the marks and the grammatical items identified by the marks. For a language for which this can be done, it may or may not be possible to capture all or many of the syntactic constraints of the language in the SGML definition.

The grammars of a very large class of languages can be defined using SGML. Even so, it is not useful to call any such language SGML unless an SGML parser is being used in its processing. Even the convention of calling SGML-defined text markup languages, SGML, makes the use of the term so unspecific as to limit its usefulness, and makes it difficult to distinguish SGML itself, the definition of text markup languages using SGML, and the text markup languages defined using SGML, because the term SGML is commonly used in reference to all three.

Another consequence of the confusion about angle-bracket languages is the misconception that SGML-defined languages are limited to angle-bracket languages. By defining appropriate sets of marks using the SGML Declaration, a wide variety of languages can be defined. In particular, it is possible to write SGML definitions for the grammars of many existing text formatting languages. SGML can be used to define the grammar of a language:

1. that uses marks for both the start and end of some elements,
2. that uses marks for only the start of some elements, and
3. that uses marks for only the end of other elements.

An SGML-defined grammar can also imply the existence of elements from surrounding elements, and allows the same mark to be used for different purposes in different contexts. These capabilities contribute to the flexibility of SGML as a vehicle for defining markup languages, and makes using the term "SGML" to describe angle-bracket languages alone even more of an impediment to understanding its use.

2.6.2.2. What is an SGML Document? An SGML document entity is an optional SGML Declaration, a document prolog and a document instance. At least a document prolog and a document instance must be present. The SGML Standard allows the system on which a document is being processed to provide an SGML Declaration, but also requires that an SGML Declaration be included in an SGML document entity if it is transferred between computer systems. In other words, any document sent from one computer system to another that does not include all three of an SGML Declaration, a prolog and

[10] Two examples are formatting languages used by Frame (see *Maker Interchange Format (MIF) Reference Manual.* San Jose: Frame Technology Corporation) and Xerox (see *Xerox Integrated Composition System — Reference Manual.* Xerox Corporation.).

an instance, is not a valid SGML document entity as defined by the SGML Standard.

In practice, it is often convenient to keep each of these parts separate when creating or processing SGML documents. For example, it is common for a number of different SGML-defined markup languages to use the same SGML Declaration, and therefore convenient not to have a separate copy of the SGML Declaration in each document. It is even more common for many documents to use one markup language. In fact, where there are very many documents, it is impractical not to do so. Keeping the document prolog separate from the document instance, and keeping only one copy of the document prolog for a set of document instances, is common and practical, for a number of reasons:

- When the markup language is modified or enhanced, only one copy of its SGML definition need be updated.
- Where there are facilities for preprocessing an SGML Declaration and prolog, so as to avoid parsing and interpreting them each time a document instance is to be parsed, a single copy of the "compiled" SGML Declaration and prolog can be used for parsing many document instances.

The way in which SGML documents are commonly handled, especially when they are initially prepared, has led to some confusion about what an SGML document really is, and has led to so-called SGML documents' being transmitted without their SGML Declaration or prolog, or with the prolog transmitted separately. Doing so, in turn, causes confusion for the recipient, especially if that person is also unclear about what an SGML document is. Two installations can legitimately exchange parts of SGML documents when doing so is convenient, but both parties must understand whether or not a conforming SGML document is being transmitted, and what that means, for the sender to be able to unambiguously transmit what the receiver expects.

2.6.2.3. What is a Valid SGML Document? Another common misconception is that if the markup in a document instance conforms to the grammar defined in the document's prolog, then the document is "SGML". This view ignores the parts of the markup language's syntax that are not LL(1) context-free, and which therefore cannot be checked by reference to the grammar in the prolog, and it ignores the semantics of the language. A marked-up document may be completely meaningless and not violate the grammar defined in the prolog (e.g. it is possible for a grammatically correct document to have all its paragraphs empty of text). Again by analogy with human languages, this is like considering the sentence "rain are thoughtful" to be acceptable, when it is not only incorrect grammatically (in a way that a straight-forward grammar will not detect: another choice for the noun subject would produce a correct sentence), but cannot be interpreted meaningfully.

This misconception has resulted in considerable confusion about the validity of SGML documents, and the whole subject of document validation. The key role played by verification and validation of text markup languages, their definitions, and documents that use them, is discussed later in this report.

2.6.2.4. What is an SGML-Defined Markup Language? As described earlier, SGML is a tool for defining the grammar of text markup languages, but the grammar of a language is only one part, and often a small part, of the language. Confusion over this point makes the statement "SGML is used to define markup languages" often incorrect, in the sense in which it is understood.

The grammar of a simple text formatting language, which consists of marks for headings and paragraphs, for left, right, and centered alignment of text, for setting the size of text, and for boldfaced and italicized text, can easily be defined using SGML. The fact that the language captures only text formatting information does not make it any less "SGML" than any other language. On the other hand, SGML was developed as a flexible method of defining languages in order to ease the implementation of text markup languages for purposes other than capturing just text formatting information. A language that, for example, allows the specification of type size with each paragraph is not necessarily "bad SGML", but, because SGML can be used to define more than just text formatting languages, such a feature is inappropriate for many SGML-defined languages, and many languages which contain such capabilities are "bad" markup languages.

2.6.2.5. SGML and Text. SGML is not just for text. Although an SGML document instance consists entirely of text characters, some of which are the marks of a text markup language, and some of which are the text being marked up, text and marks can be used to contain other than just textual information. The primary reasons for this are that many things can be represented by the type of hierarchical structure of a text markup language, and that text can be used to represent much more than human speech (in fact, almost anything). SGML-defined markup languages have been defined for capturing the information in a desktop publishing system's "style sheet", for representing music, and for representing the information required to typeset the characters in a printer's font with the correct spacing and placement of characters.

The fact that some information is not text, in the usual sense, is not a reason for not using SGML. Reasons for not using SGML are:

1. A binary encoding is more appropriate. If humans are never, in other than exceptional circumstances, going to see marked-up information, and if a binary encoding can be devised that is either easy to parse or provides the benefit of a far more compact representation than would be provided by a clear-text encoding, then a binary encoding is the appropriate choice for a language, and definition techniques other than

SGML should be used. This is usually the case, for example, for encodings of graphical information, in raster or vector form.

2. The information is so simple that the use of SGML would add unnecessary work. A simple association dictionary, which, for example, pairs SGML formal public identifiers with their system-specific counterparts, is too simple an application to justify the even the trouble of creating an SGML definition.

3. A language is very widely and commonly used, and is very stable in its definition. In this case it is appropriate and economically justified to define the language's grammar with features that are clumsy to define using SGML, to include constraints in the grammar that must be left to the application when using SGML, and to either implement a language-specific parser or use grammar definition and parser generation tools other than SGML. This set of conditions describes commonly-used computer programming languages: it is inappropriate to use an SGML parser to parse a "C" program, for example.

   Occasionally confusion arises over why SGML parsers are not implemented using such grammar definition and parser generation tools as "Lex" and "Yacc".[11] The chief reason is that, because an SGML parser is itself a parser generator, another parser generator is often not the best tool for its implementation. Another reason is that "Lex" and "Yacc", in particular, were designed, almost 20 years ago, for implementing certain kinds of computer programming languages and operating system script languages, and so are not the appropriate tool for implementing many other types of languages, in particular, text markup languages.

4. A language has a grammar so complex that SGML (along with most other computer-based grammar definition tools) is simply inadequate. This category consists primarily of human languages.

As can be seen, the use of SGML does not really depend on whether the information to be marked up is text, but rather on the complexity of the required encoding, and on the use to which the information is to be put.

A significant advantage of the ability to encode non-textual information using a text markup language is that text documents often contain things other than text. A lot of non-textual information is usually an indication that some other technique should be used, but where non-textual information is heavily interspersed with text, and where a human-usable representation for

[11] These tools are described in "Language Development Tools," by S.C. Johnson and M.E. Lesk, in *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, pp. 2155-2175. American Telephone and Telegraph Company (July-August 1978)

the non-textual information can be devised, it is often most appropriate to keep the two kinds of information together.

2.6.3. The Role of Processing Software

As was previously pointed out, an application, or a non-SGML definition of the semantics (and often some of the syntax) of a text markup language is required to make the definition of the language complete. In practice, because there is so much potential information in text, an application's interpretation of a parsed document usually only makes use of part of the available information, and so is only a partial definition of its semantics. Similarly, practicality dictates that an English-language definition of the semantics of a document type be limited to those aspects of the semantics that are of interest to the expected processing of the documents. There will tend to be a separate part of the definition for each expected processing requirement.

Many document types have a potential for being processed in a manner particular to that document type. For example, the heart of a Fault Isolation or Fault Reporting Manual[12] is a set of procedures for isolating or reporting mechanical faults found in the type of equipment to which the manual applies. An excellent use of this kind of information would be to have a computer program take a human operator through a procedure, with the human answering questions, and the computer creating whatever forms and reports are required. As well, the ability of the computer to reliably log the isolation or reporting process would be an invaluable aid in reviewing and assuring the quality of such work.

The information required to take an operator through a procedure, asking questions, and at each stage choosing the next step based on the operator's answers is quite a bit different from the information required to satisfy other requirements of a document. It can reasonably be expected that the following uses will need to be made of a Fault Isolation or Fault Reporting Manual:

1. A document will have to exist in print format to satisfy the needs of non-computerized installations.
2. The information in the document will be used in an interactive "expert system" environment, as described above.
3. A document will need to be subject to enquiries as part of an on-line data base, for example, is enquiries as to which fault isolation procedures require using a certain piece of test equipment.

Data base enquiries typically treat a large set of documents in a "global" manner: all are looked at at once. In contrast, "expert system" interaction moves through the data base in a directed manner. The semantics of how information is accessed is very different in each case. Fault Isolation and Fault

[12] The military publishing specification defining these manuals, MIL-M-83495, is discussed later in this report.

Reporting Manuals are just one example of the increasing need to be able to process information in application-specific ways.

The problem of many different semantics is made worse by the existence of many different ways of expressing even one semantic. Each desktop publishing system has its own model of how formatted text is structured, and instructions as to how text is to look when displayed have to be expressed in terms of this model. For example, some systems structure text in a hierarchical form with blocks of text at the "leaves" of the structure tree, some systems use a hierarchy in which blocks of text must also exist at all of the nodes (branch points) of the tree, and some systems structure text by placing blocks adjacent to each other, with the concept of a hierarchy replaced by constraints on the association between adjacent blocks. The problem becomes even more difficult when considering data base systems, where the semantics are not all print-formatting-oriented, and where there is therefore a wider range of possibilities.

The problem of expressing print formatting semantics in a uniform way is being addressed in a number of ways, some within the CALS initiative, some on its fringes and some completely outside.[13] Efforts are underway to develop a uniform way of expressing the semantics of "hypertext" data base systems, in which arbitrary pieces of text can be linked in a variety of ways, and to encode hierarchically structured documents prepared for use in these systems, as well as for "time-based" information for "hypermedia" documents.[14] Work is only starting on integrating the type of information required by these two classes of applications with that required for the on-line enquiry of hierarchically structured text data bases.[15]

The variety of interpretations that can apply to just one document, the variety of ways of expressing those interpretations, and the fact that many of those interpretations are particular to one or a small class of document types, makes a single, uniform method of expressing the semantics of documents a very difficult thing. Faced with rapid change, this uniformity could be an impossible thing to achieve. On the other hand, especially for documents that undergo regular revision, it is impractical to keep a separate copy of each document's text for each distinct use made of it. As much effort as possible, therefore, must be put into unifying the expression of the semantic requirements of technical documents.

---

[13] Two approaches to this problem, DSSSL and the MIL-M-28001 Output Specification are discussed in a later chapter.

[14] See Committee Draft International Standard 10743: *Information Technology — Standard Music Description Language (SMDL) and Committee Draft International Standard 10744:*Information Technology — Hypermedia/Time-based Structuring Language (HyTime) ISO: International Organization for Standardization, 1991.

[15] An important example is SFQL, refered to elsewhere.

2.7. One Markup Language or Many?

In an earlier section it was pointed out that if a language is highly stable in its design, and is widely and often used, then it is often economical to build tools for processing that language without regard to existing standards for defining grammars or generating parsers. When looking for the best markup language or markup languages for use in the CALS environment, and in light of the fact that such markup languages are sure to get widespread use, this observation raises the question, whether the top priority should be given to developing a single, highly stable, text markup language for all technical documents in the Armed Forces. Possible choices are a single language, a single language with minor allowed variations to allow for varied requirements, or a small number of languages, in order to ensure stability and widespread use. This was, in fact, the approach taken at the start of the CALS initiative, as will be discussed in a later section.

The experience of CALS was, basically, that there are too many different types of information in all the different types of technical documents used in the Armed Forces. A single markup language for all types of documents, or even a small number, would be unmanageably large, and as new types of documents are continually being developed it would be difficult, if not impossible, to maintain any sort of stability. As well, as the requirements for on-line computer access to information becomes more sophisticated, and the type and amount of information that needs to marked up in documents changes, further compromising any stability that may be possible.

The attempt to develop a "universal" text markup language ran into many of the same problems that "universal" computer programming languages encountered. Any attempt at a "universal" language turned out to be too limited in scope to address the requirements of most types of technical documents. Attempts to extend "universal" languages to encompass new requirements resulted in clumsy and unusable languages.

Ideally, each type of document would, in fact, have its own markup language, designed to efficiently and conveniently capture the type of information that those documents contain. This would address the issue that text contains a very large amount of information, only a small part of which can be directly represented by markup. This means that it is only economical to capture information that can be directly used by currently existing tools, or tools that can reasonably be expected within the lifetime of the marked-up data. As new, and often unexpected, tools develop, there is a demand for the markup of more information, introducing changes that often change the most economical way of capturing already marked-up information.

Technical document types often contain information characteristic of the type, so there is a strong motivation for taking the approach that each type has its own text markup language. On the other hand, there are so many types of documents, and it so often happens that an individual document is a type to

itself, that the resources to develop so many markup languages simply do not exist. As well, each type would require some, often significant, expenditure of effort to support formatting documents for printing, and accessing documents using an on-line data base system. Some compromise therefore needs to be struck between the two extremes of a single "universal" text markup language, and a galaxy of markup languages. As will be shown in the next chapter, it is towards this compromise that SGML usage within the CALS initiative is moving.

3. The Evolution of SGML Usage in the CALS Initiative

This chapter discusses the different components of CALS that relate to the use of SGML in light of the history and capabilities of text markup languages.

A major component of the Computer-Aided Acquisition and Logistics Support Initiative of the United States Department of Defence (CALS)[16] is the development of text markup languages for the capture of textual information for the purpose of vendor- and system-independent delivery. The CALS Initiative has provided a major impetus to the evolution of markup language design and implementation. This evolution is still very much in progress. This chapter provides an overview of text markup languages in the CALS environment, and positions CALS within the more general development of this technology.

3.1. Technical Documents and CALS

3.1.1. Publishing Specifications

Traditionally, the primary technology used for disseminating the information found in technical manuals was print, and as a consequence, the print formatting specifications for the different types of manuals still provide the largest single source of information about the document format and content. The current general specification for SGML use in the U.S. Armed Forces is MIL-M-28001, which in turn is based on the publishing specification MIL-M-38784B[17] Some of the other publishing specifications are based on MIL-M-3878.[18]

MIL-M-38784 and other publishing specifications do not specify so much what must appear in a printed document (although there are some basic guidelines for this) as how to display certain kinds of information if they are present. A lot of decisions on minor formatting details were left to the users of the specification, who were expected to be experts in the style and layout of technical manuals. Nonetheless, the publishing specifications are the primary authority on the component information present in each type of document and the interrelationship between these components. The goal of these publishing specifications is not to provide this information, and it is generally

---

[16] For overview of the CALS program, refer to Military Handbook MIL-HDBK-59: *Computer Aided Acquisition and Logistics and Support (CALS) Program Implementation Guide.* 20 December 1988

[17] Military Specification *MIL-M-38784 — Technical Manuals: General Style and Format Requirements.* (16 April 1983).

[18] An example is Military Specification MIL-M-83495: *Manuals, Technical, On Equipment Set, Organizational Manuals: Detailed Requirements for Preparation of (For USAF Equipment).*

not possible to design a text markup language (other than a text formatting language, and then only in part) based entirely on these specifications.

The limitations of a publishing specification become clear when an attempt is made to produce a formal description of a markup language for the class of documents based on the publishing specification. Often there is not enough information explicitly stated in the specification to determine what combinations of structural elements are allowed in conjunction with others, and what elements exclude the presence of others.

Because publishing specifications do not provide all the information needed to develop a text markup language, directives must also be provided indicating the goal or goals of the markup language being designed. At its simplest, the goal could be to reproduce the print forms of the documents. At its most complex, the goals could be to provide enough information to enable an interactive multimedia presentation of the documents to be created. The driving goals of the CALS initiative are to take operational documentation away from a solely print-oriented technology, so there must be stated goals for any markup language project that take it beyond simply reproducing print documents.

3.1.2. An Early View of SGML

In the early 1980's most text markup languages used in the publishing and printing community consisted of a fixed set of "tags", each with a fixed, defined meaning. Some such markup languages provided a "macro" facility, so that new tags could be defined, but these new tags were almost invariably minor variations of existing tags, and so did not change the nature of a markup language. These markup languages were almost all publishing-oriented and had semantics that were well understood in terms of print formatting.

SGML was developed, in part, to sever this strong link between markup tags and print formatting semantics. It provides the means for describing the grammar of a markup language, and associates tags with the components of that grammar, without any reference to the tags' semantics. The development of SGML was partially motivated by the anticipated increased use of text in on-line data bases, and partially motivated by the need to prepare, deliver and revise print documents without being tied to a single processing technology. At the time, there was little familiarity with data base use of text (a situation which is only slowly changing), so the emphasis continued to be on print technology. CALS anticipated data base use but there were no specific provisions for data base use.

3.1.2.1. MIL-M-28001. The emphasis on print technology meant that the original CALS use of SGML was heavily print-oriented. The first specification

for SGML use, MIL-M-28001,[19] contained a "conforming" DTD, defining a "Universal" markup language for technical manuals.

The text markup language definition in MIL-M-28001 is based on an SGML DTD, and a "Data Dictionary" that describes the function, typically in a few words, of each tagged element. There is no description of the markup language or of the information it captures in non-SGML terms. There is no indication of what requirements each component of the text markup language is intended to satisfy.

Early in the development of MIL-M-28001, an appreciation developed of the wide variety of print formatting requirements in technical manuals. MIL-M-28001 supported this by providing a "non-conforming" markup language on which the development of "variant" markup languages could be based. A study of markup languages developed to support just one publishing specification, MIL-M-83495, found that some MIL-M-83495-based markup languages could appropriately be made very similar to the "conforming" markup language of MIL-M-28001, where others had requirements closer to those being addressed by the HyTime language being developed for "Hypermedia" documents.[20] The same study also found that different solutions were applied when solving the same problem of how to capture a certain type of information, even when the document types were closely related, and both solutions were produced by the same contractor.

The experience with MIL-M-28001 was that all markup languages used in practice were variants. The "Universal" markup language encountered a similar fate to that of "universal" computer programming languages.

The lack of guidelines for developing variant markup languages has lead to examples of arbitrary variations in markup languages intended for use with closely related documents. This difficulty has not been addressed by MIL-M-28001A, and is the most important sub-component of text markup language use within the CALS initiative as yet undeveloped.

3.2. "C"-Type Documents

The inadequacies and inflexibility of MIL-M-28001, especially with regard to using documents in a data base context, lead to revision of some of the original requirements. A new revision of MIL-M-38784 was developed.[21] Variant markup languages were developed incorporating general

---

[19] Military Specification MIL-M-28001: *Markup Requirements and Generic Style Specification for Electronic Printed Output and Exchange of Text.* 26 February 1988.

[20] *CALS DTD Verification and Validation Final Report,* F33600-89-D-0164, written for the U.S. Air Force by Software Exoterica Corp. under contract to Century Technologies, Inc. 1990

[21] Military Specification *MIL-M-38784C — Technical Manuals: General Style and Format Requirements.*

improvements to MIL-M-28001. As well, revisions to MIL-M-28001 were developed, culminating in the publication of MIL-M-28001A.[22]

3.2.1. The Development of MIL-M-28001A

MIL-M-28001A incorporates better documentation for the tag set, less emphasis on the "conforming" markup language, and an "Output Specification" language for specifying print formatting semantics. The primary emphasis of the SGML definition is towards providing a pool of standard elements and structures (the "baseline tag set") that can be combined together to form a text markup language. A "conforming" language is still provided, but only as an example of using the baseline tag set.

MIL-M-28001A is still, however, fundamentally similar in structure and form to MIL-M-28001. The definition of the markup language is still oriented towards the SGML definition of the markup language. There are no guidelines for how to use the baseline tag set or how to develop new text markup languages. The guidelines provided are a short, useful, tutorial on the general use of SGML.

3.2.2. Document Processing — The Output Specification

MIL-M-28001 included a specification for print formatting each of the elements defined in the "conforming" text markup language.

The specification language itself, called the "Output Specification" or OS, is itself a markup language whose grammar is defined by an SGML DTD. A marked-up document (i.e. a document instance) using the OS language is called a "Formatted Output Specification Instance" (FOSI). Corresponding to each element in the "conforming" markup language in MIL-M-28001 there are one or more structures in its FOSI that specify print formatting characteristics. The Output Specification is, in effect, a simple text formatting language, that binds the tag set of MIL-M-28001 to print formatting semantics.

MIL-M-28001A developed the idea of the Output Specification further. In addition to structures specifying print formatting for individual elements, the MIL-M-28001A FOSI contains structures that define overall characteristics of a printed document, such as page layout and page headers and footers. The formatting language used in MIL-M-28001A is functionally much more complete than that of MIL-M-28001, and its components are more completely defined.

Until the use of OS-based text formatting systems has been demonstrated, it will be hard to evaluate the effectiveness of the FOSI approach to specifying print formatting. The incompleteness of the MIL-M-28001 OS language inhibited its use, but the near future should see one or more MIL-M-28001A

---

[22] Military Specification MIL-M-28001A: *Markup Requirements and Generic Style Specification for Electronic Printed Output and Exchange of Text*. 20 July 1990.

OS language-based "FOSI compilers". If not, their absence will indicate that further effort is required in this direction.

The Document Style Semantics and Specification Language (DSSSL), currently under review as a draft international standard by ISO,[23] is a more general approach to associating processing semantics with the structure captured by text markup languages. It provides a text and structure processing language together with a specification mechanism for print formatting semantics. Its intent is more general than that of the Output Specification, in that it is intended to be used for associating data base semantics with document structures as well as print formatting semantics. Evaluation of its effectiveness, like that of the Output Specification, has to await demonstration of its use in an application of significant size, such as a full technical manual.

[23] ISO/IEC DIS 10179: *Information technology — Text and office systems — Document Style Semantics and Specification Language (DSSSL)*. ISO: International Organization for Standardization, 1991.

4. The Road from Here to There

The previous chapters have presented the basic problem faced by CALS: the multitude of divergent requirements for capturing the information content of text documents. This chapter elaborates on the issues that have to be addressed in solving this problem, and describes a methodology and set of tools that can be used in the solution.

4.1. The Problem — A Multitude of Text Markup Languages

As has been seen, the major problem faced by CALS is the multitude of text markup languages under development, and soon to be developed, with very little control over how those markup languages are to be developed.

MIL-M-28001A provides a "baseline tag set" with definitions for the elements identified by those tags, formatting information for the elements, some introductory material on the use of SGML, and a sample "conforming" DTD for a markup language using the baseline tag set,. There are a few short guidelines for developing new markup and modifying the existing markup, and nothing about the philosophy or justification used in the development of the baseline tag set. MIL-M-28001A documents the SGML definition of the markup language represented by the baseline tag set, rather than documenting either the markup language itself or the structure and information captured by the markup language.

Tri-Service use of MIL-M-28001A is inhibited by the lack of a uniform way of adapting MIL-M-28001A to the needs of the individual services, or to the variety of requirements within each service.

4.1.1. Addressing the Right Audience

For a text markup language to be usable, its use must be fully documented. Users of the markup language, and of the information captured by it, have to be able to learn about the markup language in terms they understand. There is more than one kind of user of a text markup language. Each kind of user of a markup language needs a different kind of documentation.

A useful analogy can be drawn between the documentation required for a text markup language and that required for a piece of equipment: a vehicle for example. Four types of documentation are required:

1. Operational manuals, describing how to use a markup language.
2. Interface manuals, describing the place and function of a markup language in a computer system which is to prepare and process marked up documents.
3. Maintenance manuals, describing how to modify a markup language.
4. Design manuals, describing the principles on which a markup language was designed and the requirements it is designed to satisfy.

Following is an outline and description of the requirements of each of these types of supporting documentation.

4.1.1.1. Marking Up Documents. Personnel who actually markup documents need to be provided with a description of the mark up languages they are using that deals with the format of marked-up documents. Starting a new paragraph on a new line with an identifying piece of markup in a marked-up document is as much a formatting issue as is indenting a paragraph and surrounding it with space in a printed document. The reasons for choosing a format are similar: in each case information is displayed in a fashion most comprehensible to personnel reading it. Issues that have to be dealt with include:

1. How to recognize each type of thing that is to be marked up. For example, how to recognize a text paragraph and distinguish it from a title or verbatim example.

   When marking up documents that have already been printed, individual items are most often identified by their print format: paragraphs are indented, titles are centered and in bold-face type. Items in newly marked-up documents, on the other hand, need to be identified in a manner that depends on who is doing the markup. Items can be identified by their content if the markup personnel are familiar with the subject matter of the documents being marked up. If they are not, some sort of pre-markup is required to identify textual elements to markup personnel. The exact form of these marks depends on the documents and personnel. Marks similar to traditional copy marks would be suitable for documents that are primarily to be marked up using print formatting information.

2. What marks identify each type of thing and where they are put. A paragraph, for example, may be marked up using a tag at the start of the paragraph.

   In some cases it is appropriate to provide different markup for an SGML element in different contexts. For example, when using MIL-M-28001, it was necessary to place markup immediately following the end of a footnote, without any intervening space or line breaks. Either the end of a surrounding element that would normally not be required, or the starting markup of the following element on the same line as the end of the footnote, was required. In more advanced text markup languages, the context of an element may determine how it is marked up. For example, a keyword would have to be surrounded by identifying marks when used in normal text, but could be marked up differently where it was defined as a term, a context in which list-oriented markup would be appropriate, and for example, the markup for the definition of a term would provide the end of the term.

Where the type of markup varies by context, the variations need to be documented, either as exceptions to the "usual" markup, or as different pieces of markup. In the latter case it may be appropriate to treat an element as being a different "thing" in each context.

It is not appropriate to describe markup in SGML terms. Markup operators are typically not interested in whether a piece of markup is a tag or a short reference. They are not interested in whether or not the OMITTAG feature is being used, but are interested in the effect that using SGML's features has on marking up documents.

3. Other issues of formatting marked up documents need to be dealt with, especially regarding the use of white-space in marked-up documents. Such issues include how many space characters go at the end of a sentence, where spaces and line breaks need to be placed, or must not be placed around pieces of markup, and the use of blank lines. Rules need to be stated about the symbolic names assigned to sections and paragraphs when these are used to cross-link references within documents.

In general, personnel who actually mark up documents only need to know enough about a document to recognize its components. Ideally, they need not even be aware of the existence of SGML. Their job is to mark up a document using a particular markup language provided for that document. For this type of user, all they see of SGML is the tag set and other markup specified by the DTD that defines the markup language they are using.

Although in most cases not required, there are often reasons markup personnel should have some knowledge of the principles behind the markup language they are using. Such knowledge helps markup operators recognize and deal with situations in which the data that they are entering does not fit with the markup language they are using. Especially when pre-existing documents are being marked up, and when these documents vary from the standards and guidelines on which the text markup language is based, decisions have to be made that have the potential to change at least the appearance of the marked-up document when, for example, it is next printed. When such a variance is discovered, there are usually two choices available for how to mark the variant text:

- The text can be modified to conform to the markup language. This often consists of something as simple as changing the manner in which lists and sections are marked or numbered. In other cases, the sectional structure of the document may need reorganization: sections may become chapters, for example, or new titles may need to be added.
- The text can be marked up in the manner that causes it, when next printed, to as closely as possible resemble the way it did when last printed. For example, a paragraph with a run-in bold-face title could be

marked up as a paragraph with no title but with bold-face emphasis of text at the start of the paragraph.

The first of these two methods is always preferable, but is usually the more difficult to put into practice. The difficulty is produced by the fact that some authority is usually required to modify the appearance of a document, and markup operators often do not have this authority, nor do they have ready access to those who do. Existing standards and guidelines are almost all print-oriented, as is the experience of markup and production personnel. This means that there are typically few personnel who are familiar with the new requirements that are the reason for using a MIL-M-28001A-based text markup language. Often the personnel with the authority to make decisions concerning the appearance of a document do not have this familiarity, making the situation even more difficult.

As a consequence of these difficulties, the second method, preserving the appearance of the printed document, is most often seen in practice, even though it causes information to be lost. If titles are not identified, for example, a complete table of contents cannot be produced. This may not be of significance for the printed document, where some titles or headings may be omitted in the print document, but it could be significant for future uses of the document, where these titles may be required for an on-line subject index.

4.1.1.2. Processing Documents. Personnel responsible for processing marked-up documents are generally not interested in the exact manner in which they are marked up. The ready availability of tools such as SGML parsers means that the computer software that does the processing (formatting, conversion, loading into a data base) need only deal with the hierarchical structure of a marked-up document, and with its text. The marks used, and whether a grammatical item was preceded, followed or surrounded by marks, do not affect the processing.

The grammatical structures identified by the text markup language are directly available to the the processing software. Issues that have to be dealt with in documentation directed towards processing a markup language address that structure, and the text embedded within that structure:

- The function, in terms of capturing information, of each element has to be documented.
- All grammatical issues not dealt with by the SGML definition of the markup language have to be documented. The significance of leading, trailing and embedded spaces, and the meaning of line breaks has to be made explicit.

Personnel who implement computer programs that process marked-up documents typically need not be SGML experts, and the documentation of markup languages they use should reflect this. They are similar in this regard to personnel who mark up documents. On the other hand, unlike markup operators, they must be very much aware of the nature of the content of

processed documents. These requirements are simple, and of a sort familiar to markup operators, when the requirements refer to the processing of print formatting information. The requirements are somewhat more complex regarding the type of information that is dealt with by data base systems.

4.1.1.3. Maintaining a Markup Language. Once a text markup language has been designed it will probably require regular maintenance as user needs and the supporting technology change. Well designed markup languages should anticipate this change to some extent, reducing the required maintenance. However, the changing approach of the CALS initiative since its inception to the design of text markup languages is ample evidence that such change is the norm rather than the exception. Maintenance of text markup languages, like the maintenance of anything else, is based on the availability of supporting documentation.

SGML expertise is required of the personnel who create and maintain a text markup language. Documentation directed at this group of personnel can reflect this expectation. The designers and maintainers of a markup language, however, have to be constantly aware that it is designed to be used by other than its designers. As a consequence, any modification of a markup language must refer to the documentation prepared for markup operators and implementors of processing software.

When modifying a text markup language, it is important that changes have a minimal impact on prior investment in both marked-up documents and processing software. The following issues have to be considered:

1. It is often prohibitively expensive to modify documents that have already been marked up and which are not due for revision in the near future. Preserving this investment has to be the top priority when any change to a markup language, or to the systems that process it, is being considered.

   It was practical, when the only use of computer-stored documents was to print them, to have different documents of the same type marked up in different fashions. The only requirement for preserving the investment was to use the computer software designed for each markup language when processing documents marked up using that language. With the expected advent of highly structured and integrated textual data base systems, all documents must be available for processing by the same computer software, and the approach of using different pieces of computer software is no longer possible.

   The most straight-forward way of preserving an investment in existing markup is to make any change in a text markup language "upward compatible": existing markup must be still valid in the modified language. The modified markup language differs from the old one in having additional types of markup, all of which are optional: available for use in new documents, but not required in existing documents.

The only other practical way of dealing with change when there are a large number of existing documents is to "convert" the existing documents. Well designed text markup languages readily lend themselves to this sort of processing, the reason being that they are designed to be processed by computer software for a large variety of reasons. However, especially where documents have been archived, such conversion may be prohibitively expensive.

2. Computer software is often very expensive to modify, even when only one computer system is involved. As has been noted above, as the CALS initiative progresses, there will be many computer software systems potentially impacted by changes to text markup languages they process, and the cost of change can be expected to increase in the future.

   Modifications to text markup languages require processing software to be changed. The amount of change can be minimized if changes to a markup language are upward compatible. Both old and new documents must be processed by computer software after an upward compatible change, but so long as the text and structure captured by the new markup can either be ignored, or the text processed while the new structure is ignored, the requirements of the new documents are minimal.

3. Markup personnel vary considerably in their ability to adapt to new or modified text markup languages. Upward compatible changes, which have a minimal impact on what such personnel already know, are easiest to introduce into the workplace.

   The cost to the markup process of changing a text markup language must not be underestimated. It is often the largest component of the cost of doing so. The cost is incurred by training, by reduced productivity while personnel are adapting to changes, and by the potential for increased incidence of errors in marked-up documents. The cost of errors in marked-up documents is greater than is usually recognized, because it is the only cost of those mentioned here that is realized only when use is made of the final product resulting from processing marked-up documents, and is therefore deferred. The other costs are more immediately apparent.

In spite of the large cost of change, provision must be made for it in planning for Tri-Service use of SGML: the only alternative is to abandon the advantages of a rapidly progressing technology. A strategy needs to be developed that ensures that text markup languages developed for Tri-Service use can be maintained in an upward compatible manner, and that computer software developed to process these languages be easily adapted to upward compatible changes.

In addition to a general strategy for text markup language maintenance, each language must be supported by maintenance documentation that addresses the issues raised above in terms of the particular language. A "maintenance

manual" is as important for a text markup language as it is for a piece of equipment.

4.1.1.4. Designing a Markup Language. New text markup languages will be required as new classes of documents are encountered in the process of converting to system-independent document management technology from print-based or system-specific technology. There are three major considerations when developing a new markup language:

1. There must be good reason for developing a new markup language in preference to using an existing one. Documentation must be available for existing text markup languages that clearly describes the requirements these languages satisfy and the principles used in their design. This documentation can be used to efficiently and accurately determine to what extent the goals and requirements of existing markup languages match or differ from the requirements of newly encountered documents.

2. The new language must fit into existing processing technology to as great an extent as possible. The kinds of structure captured by a new markup language must match that captured by existing markup languages as much as is possible if existing processing software is to be used with a minimum of modification. A well designed markup language will have a consistent basis for its structures. The same kind of things, for example titles and headings, will have the same kind of structure and the same options on what is allowed within it. Processing for similar types of things will therefore be similar.

   An example of the need for uniformity of structure is where a marked up document is to be formatted for printing. The fewer basic types of structures used, the less specification has to be provided to describe the distinct formatting of each structure. For example, if lists are allowed either to be embedded within paragraphs or to appear between paragraphs, different processing logic will normally be required in each case. The documents being marked up may demand this distinction, but the cost of supporting one of two such forms when only the other is already supported needs to be considered in the designing of a new text markup language.

   Design documentation should describe the existing kinds of structures as a basis for making these kinds of design decisions. Such documentation also allows decisions to be made about the applicability of existing or new processing software to documents, especially when, as is presently the case with most text management and display software, the ability of the computer software to manipulate arbitrary data structures is very limited. In some extreme cases it may be appropriate to design new text markup languages with the specific purpose of supporting some very limited computer software. In this case, study of the design documentation of existing markup languages is critical to determining

exactly what requirements have to be made of the new markup language to support the software's limitations.

3. The marks used by new text markup languages must be as consistent as possible with those of existing text markup languages. A review of the design documentation of existing text markup languages must be preformed to find those ways in which the new language can be made to most closely resemble the existing ones. The way in which names used in marks are chosen should be uniform, and the same name should be used for the same structure (e.g. a title should use the same mark). If SGML short references are used to mark the items of lists and to mark table entries, all documents used in an organization should use the same short references in the same way.

Designing a new text markup language is very much like modifying an existing one. The same considerations of preserving existing investments apply in both cases.

There is a reason for documenting the design of a text markup language other than using the documentation to aid the design of later markup languages. A text markup language, like a newly designed piece of equipment or computer program, has to be subject to a quality assurance process. The record of the design process is the primary subject of the independent verification and validation that confirms the quality and appropriateness of the text markup language. The design documentation for a markup language, therefore must contain:

1. A list of all the requirements that formed the basis of the design, together with a report on the extent to which the markup language satisfies each requirement.
2. A report of how each requirement in any pre-existing specification (such as a publishing specification) is met by the markup language.
3. A description of the principles on which the markup language was designed, such as naming conventions, etc.

4.1.2. The Impact of Processing Technology

The current state of SGML usage, both in the Armed Forces, and in the private sector, has been largely determined by the methodology and the tools currently available for creating and processing both markup languages and text documents.

4.1.2.1. Methodology. A methodology is a way of doing something. In the case of designing a text markup language, the methodologies of interest are:

1. how to determine what information in a set of documents needs to be captured,
2. how best to structure the captured information for use by processing applications,

3. how best to mark up the documents in order to capture the information with the desired structure (i.e. how to design a text markup language for the information in the documents), and

4. how to define the resulting text markup language using SGML.

In addition, processing methodologies are required for the use of text markup languages. Processing methodologies of interest include:

- how to create and revise marked-up documents,
- how to produce useful products (e.g. readable pages, searchable on-line documents) from marked-up documents.

Examples of two different processing methodologies are text-markup language-based-preparation of text using a classical text editor and WYSIWYG ("What You See Is What You Get"). Processing methodologies depend for their effectiveness on the tools available to implement them. For example, prior to the existence of textual data bases there was no use for, and no way of practically evaluating the effectiveness of, processing methodologies for data base use. A processing methodology does not imply a choice of what computer system or software products are used, but rather the capabilities of those products.

Design methodologies do not depend on high-tech tools. Pencil and paper are adequate tools. A design methodology is a way of working with pencil and paper. This difference between processing and design methodologies is the basic reason that it is practical to design text markup languages for computer applications, most of which will come into existence at some time in the future. So long as a reasonable prediction can be made about the capabilities of processing tools, and so long as these predictions include some judgement as to the kind of processing methodology that will be used with those tools, design methodologies can be developed for markup languages that anticipate these future processing methodologies.

The "semantics" captured by a text markup language is generally defined in terms of particular processing. For example, markup for a paragraph is associated with laying out the text of the paragraph as a block on a page or screen and making it look "normal" (not distinguished as a title or quotation). Paragraph semantics are "partial" in this fashion because the exact implementation of these requirements depends on the medium and tools used to display a paragraph. Processing semantics generally have to be defined in this partial manner, not only to allow for their implementation using a variety of media, but to allow anticipation of tools and techniques not yet fully understood. A processing methodology should incorporate partial specification where it is appropriate.

Semantics have to be assigned to text markup languages for their use to be meaningful. Therefore, a processing methodology, incorporating a set of semantics, generally has to be associated with any newly defined text markup

language. The semantics of the processing methodology can be partial, but something must be there. For example, it is insufficient to design a markup language for text that is intended to be used in a data base system without identifying what pieces of information are captured, and how, in general terms, they can be used.

At present there are no standard methodologies in common use. Effective Tri-Service use of text documents requires that there be developed standard text markup language design and processing methodologies.

4.1.2.2. Tools. Computer programs provide the tools for using text markup languages. Most text entry at present uses either a WYSIWYG word processor or desktop publishing system, or a classical text editor. A text editor is essentially a "glass typewriter", with formatting typically limited to a single font, type style and size, and to "spacebar" type spacing. From a WYSIWYG point of view, a text editor is just a very simple word processor, although text editors have a very important characteristic that word processors lack: text prepared using a text editor is typically easy to transfer to another computer system, be it another text editor or a processing program. Other methods of text entry are provided for hypertext systems and other data base systems, but these are usually simple variants on WYSIWYG systems or text editors.

Computerized print formatting of text has been common since the 1960's, based on either a WYSIWYG approach or on specialized text markup languages. Data base use of text is distinguished from print formatting in that useful "small scale" text data bases are not common. The whole point of putting text on a computer is that the volume of information exceeds the capabilities of traditional media (books and manuals). This is unlike the situation with print formatting where the scale of documents ranges from single memos to multi-volume encyclopedias. Standards for data base use of text are currently under development[24] and the development of systems based on these standards can be expected by the mid 1990's.

New markup languages have to accommodate existing tools. There is no point in capturing information that these, or soon-to-exist, tools cannot handle. An understanding of the available tools must form the basis of determining what kind of information new text markup languages capture. This understanding must be reflected in the design documentation for new markup languages.

4.2. The Solution — Managing Technical Documentation

The current ad hoc approach to developing text markup languages has not worked. As was found with the set of markup languages developed based on the MIL-M-83495 publishing specification, different solutions to the same

[24] *SFQL: Structured Full-Text Query Language — Specification* by Dr. Neil R. Shapiro. GE Corporate Research and Development.

markup problem will be developed if there are no strong guidelines for development.

Based on what has been learned about the management of text, Tri-Service use of SGML must be based on the formalization of design and processing methodologies for text markup languages. There will be many markup languages produced to satisfy the varying requirements of different services. Management of these languages, using guidelines and standards for developing text markup languages, is what can and must be done. The alternative, a multitude of incompatible markup languages, will eradicate most of the benefits of the CALS initiative.

Management of technical documentation, and of the way in which it is encoded, relies on the use of guidelines and standards, but this reliance must not be made on inappropriate standards or inadequate guidelines.

4.2.1. Guidelines and Standards

The term "standard", applied to SGML, has misguided those not familiar in detail with ISO 8879, the SGML standard, as to the extent its application provides a management tool for text. A similar misapprehension has arisen from viewing military publishing specifications and MIL-M-28001A as standards. Each of these documents provides tools for describing the structure of text and its format, and provide some basic guidelines for using the tools. The very generality of their specifications, which allows each of them to apply to such a wide range of documents and applications, means that specific requirements of applications have to be dealt with by other means than these standards and specifications.

Additional guidelines are needed, primarily for the production of documentation that supports the design and use of text markup languages. Once it is ensured that adequate documentation is being produced for new markup languages, more specific guidelines for standard marks and structures can be developed.

4.2.2. What Can Be Standardized?

Guidelines can be developed for standardizing many of the activities related to text markup language design. The following is a list of some of these:

1. Common print-format-oriented elements, and their associated markup. Paragraphs and the common sectional structure of body matter, together with their titles, are examples of these types of elements.
2. Common attributes, with wide utility, such as security identification. Standard attributes can be used in a new text markup language even with non-standard elements. The more uniformity there is in attribute usage, the simpler it is for both markup operators and implementors of processing computer software.

3. Conventions for the names of elements can be established, so that by their consistency they are easier to remember.

   Naming conventions have to consider the trade-offs between long and short names, the former easier to understand, the latter easier to enter. Longer names are appropriate for rarely used elements, shorter ones for commonly used elements.

4. Conventions for the identifiers used to cross-link elements when text refers, say, to a figure. Similarly, conventions need to be developed for naming external objects, such as graphics, that will make documents more nearly portable across a variety of computer systems.

5. Common markup for tables is required that deals with some of the complex formatting that commonly appears in them.

6. Conventions for the appearance of DTDs, and the names used in them, other than those used in tags and general entities, such as parameter entity names and short reference map names.

The use and limits of each of these standardized items and conventional procedures must be documented. For example, if paragraphs are standardized, something has to be said about what is a paragraph and what is not, to minimize the chance of one markup language considering a structure a paragraph, and another considering a similar structure to be not, with no justification for the difference.

It should be emphasised again that there is no use in having standard elements and attributes if their use in a text markup language is not fully documented.

4.2.3. New Methodologies

An important set of methodologies for managing a large number of text markup languages depends on the concepts of partitioning the set of languages to produce a set of markup language families, and of partitioning text markup languages into sublanguages, making it possible to reuse whole sublanguages in designing new markup languages, not just individual standardized elements ("Divide and conquer").

4.2.3.1. Text Markup Language Families. A text markup language family is a set of markup languages which have many components in common, and which vary in a uniform manner. For example, repair manuals for different classes of equipment, which are identical except that each class of equipment needs a different structure to list exceptional procedures, would form a family with each equipment class having its own member markup language. Text markup language families have the advantage that only the manner in which the members differ have to be documented separately for each member. This greatly reduces the cost of maintaining the documentation, the

cost of training for personnel marking up new family members, and the cost of implementing processing for new family members.

4.2.3.2. Markup Sublanguages. A text markup sublanguage is the markup for a set of related elements in a document structure. For example, common structures for paragraphs, sections and titles used with those sections could form a sublanguage. Documentation can be produced for sublanguages, meaning they can be incorporated in new text markup languages with virtually no overhead incurred by the cost of documentation. Each sublanguage needs to be documented as to its appropriate use, so that it is well understood when it should be used.

4.2.3.3. Common Processing Semantics. Common processing semantics are the application of the same meanings to similar syntactic constructs in different text markup languages. The reduction in the variety of processing semantics reduces the cost of implementing processing computer software. The application of common semantics requires the formal definition of these semantics on one hand, and a way of using the semantics on the other. For example, if paragraphs, sections and their titles have common processing semantics, it should be possible to take the part of a FOSI that describes how to print format them and use that part in other FOSIs. To do this requires the creation of FOSI "sub-instances" that are similar in their intent, and should often parallel, text markup language sublanguages.

Another method of applying common processing semantics is to have a single "processible" markup language that incorporates all the processing semantics for a markup language family. Processing can then be accomplished at reduced cost of implementation by converting documents marked up using family members into the "processible" family member, which is the only markup language in the family ever subject to processing. Very large families can be created to take advantage of this technique.

4.3. What Next?

Two things need to be done: the proposals of this report need to be validated, and they need to be implemented. Most of the supportive effort that has resulted in the specification for SGML usage in the Armed Forces, MIL-M-28001A, has up to now concentrated on developing specific components of text markup languages, and techniques for specifying how marked-up documents are print formatted. This emphasis needs to change towards developing guidelines for the design and use of text markup languages. These guidelines should emphasize the role of documentation in the development and support of a text markup language.

Documentation produced to support text markup language needs to take advantage of advances in CALS technical documentation technology. In particular, there needs to be developed a MIL-M-28001A-based text markup language for text markup language manuals. This markup language needs to

be subject to Independent Verification & Validation (I V & V), like any other markup language.

The proposals of this report can be validated and verified by initiating a project to design a new text markup language based on strong guidelines. Such a project would have to be preceded by development of these guidelines, but that process itself should have the advantage of introducing new tools into the arena of CALS markup language development.

Once new guidelines have been developed, existing text markup language development material should be subject to I V & V. MIL-M-28001A should be I V & V'd, as should existing markup languages and the Output Specification language. Doing this will place the CALS use of text markup languages on a much firmer footing.

4.4. Independent Verification and Validation[25]

The sequence of tasks which produces a new text markup language consists of the following steps:

1. background analysis of the standards and specifications on which the new language is based, of existing text markup languages, and of the history of past developments in related areas;
2. examination and analysis of a selected class of related documents, and of the use that is expected to be made of those documents;
3. design of a markup language which effectively and efficiently captures the information in documents of the selected type;
4. formal description of the markup language using a Document Type Definition (DTD) conforming to the Standard Generalized Markup Language (SGML) standard;
5. if required, production of a formal specification or specifications of how marked-up documents are to be processed (using the Output Specification defined in MIL-M-28001A for print formatting, if its use is specified);
6. testing the usability of the markup language and its formal description by marking up sample documents of the selected class;
7. analysis of the effectiveness of the markup language and its formal description by processing the sample documents (using the formal specification, if there is one) in ways that exemplify their expected uses; and

---

[25] The I V & V task is discussed in more detail in *How To Do Independent Verification And Validation On An SGML-Defined Markup Language,* written for the U.S. Air Force by Software Exoterica Corp. under contract to Century Technologies. 1991.

8. independent verification and validation the text markup language, its SGML definition and its supporting documentation in order to determine if the preceding tasks have been completed, how accurately and effectively each one was accomplished, and how well each task is documented.

I V & V is the final task in the process of creating a new text markup language. I V & V is crucial to confirming the usability of a markup language.

4.4.1. What is I V & V?

The I V & V task consists of subtasks that parallel the tasks that preceded it: analysis, markup language design, formal definition, and testing. Personnel doing I V & V follow a set of procedures that match those followed by the personnel who developed the markup language. At each point in each procedure there must be a document, on paper or in machine-readable form, which provides the link between the original task and the I V & V subtask.

The background analysis of the standards and specifications, of existing text markup languages, and of past developments, needs to be done both by the designers of the markup language and by I V & V personnel. This analysis provides the basis for understanding the new markup language and the way it will be used. Verification and Validation requires the same clear understanding as does the initial design.

Where possible, repeatable tasks should be performed by the I V & V personnel. For example, the sample marked-up documents should be reprocessed so that the practicality of the production process can be evaluated, and the processed forms (printed pages, on-line accessible data base) should be examined and compared to those delivered as part of the design documentation for the markup language. In some cases the I V & V personnel may not have the resources to repeat all such processing, but the one task that must always be repeated is machine analysis of the SGML definition of the markup language (the DTD) and of the sample marked up document. In other words, the I V & V personnel must parse the DTD and parse the sample documents.

The other subtasks consist of an examination and review of the documentation that was produced by the markup language design personnel. The sample documents provide important clues as to the usability of the markup language. Two kinds of errors can be made:

- Syntactic errors are violations of the constraints of the grammar of the markup language. These errors can be found by an SGML parser applied to the markup language's DTD and the sample documents. Errors of this sort should never be found, because the markup language designers must have had an SGML parser available to them to be able to develop the DTD and process the sample documents.

- Semantic errors are violations of the intended use of syntactic structures. For example, a paragraph with no text in it is almost always indicative of an error. A keyword that should be, but is not marked up, is another type of semantic error.

Detection of some semantic errors can be automated. There are three kinds of semantic errors:

1. Original sample documents may be inconsistent in format. Traditional print media typically encourage inconsistency because, prior to the introduction of the computer to publishing, there was no effective way of checking for inconsistency other than by direct examination. An example of a symptom of this type of error in a document is a title with no text inside it. This may be a result of the grammar for a section requiring a title but an original print document from which a sample was taken having no title for the section.
2. The design of the markup language may be in error because it cannot be used to capture the structure of existing documents. A title empty of text could be a symptom of this kind of error as well.
3. A sample document may be incorrectly marked up. An empty title could be a symptom of a poor choice of markup by a markup operator: there may have been a more correct form of markup that did not require a title. Incorrect markup could also be a symptom of inadequacies of the documentation produced for markup operators.

Processing the sample marked-up documents is an important step in I V & V. Even if the intended use of marked-up documents is in an on-line data base, print formatting the sample documents is a source of visual clues to problems with the markup language, as well as a way of testing its utility in a processing environment. Such processing confirms the value of the text markup language and is an efficient way of finding semantic errors in the sample marked-up documents ("a picture is worth a thousand words").

Things like titles with missing text or text that looks "wrong" (for example a title that is a number) can be detected by computer software, the use of which aids in the quality analysis of marked-up documents.[26] Other types of errors, such as missing markup, can only be detected by examining marked-up documents or the printed pages that result from processing the marked-up documents, and comparing them to the intended print result, if one is available.

[26] An example of such software is the "Semantic Analyzer", produced for the U.S. Air Force CALS Test Network by Software Exoterica Corp. under contract to Century Technologies Corp.

4.4.2. Complete Documentation

An important part of the I V & V is confirming the completeness of the documentation provided to support a text markup language. Without complete documentation, a markup language is unusable.

If the markup language project has not produced all the documentation required to do the analysis required by I V & V, the I V & V personnel may have to create the missing documentation in order to enable them to complete the I V & V task. Documentation that may have to be produced by the I V & V task includes:

- Design documents describing the text markup language and its SGML definition. These documents are the required link between the markup language and its initial requirements, and are needed for the analysis of the markup language's satisfaction of these requirements.
- Sample marked-up documents using the markup language. If sample documents are incomplete in terms of containing samples of all types of markup allowed by the markup language, or if they are missing, I V & V personnel must create sample documents.

Such documentation should be incorporated in the documentation for the text markup language, so that it becomes available to users of the markup language. The two types of documentation listed above are the minimum required to perform I V & V. If other types of documentation are missing, such as documentation for the markup operator or processing software implementor, the markup language may still be unusable.

4.4.3. What Can Be Subject to I V & V?

Until now the I V & V of the design of a text markup language has been described. Other things can be subject to I V & V:

1. Text markup language families can be subject to I V & V in much the same way as individual text markup languages. All members of a markup language family share documentation. The effectiveness of a markup language family can be tested by designing an individual markup language that fits within the family and then performing I V & V on it. If the family is well designed, adding a new member to it should not require undue effort, so this should be a practical activity.
2. Marked-up documents need to be subject to I V & V as a Quality Assurance activity. The I V & V task in this case consists of checking for syntactic and semantic errors in marked-up documents in much the same manner as for the sample documents that are part of a markup language design project. Processing the sample documents provides the additional benefit of confirming the effectiveness of the markup.

3. Processing of marked-up documents can be subject to I V & V. Sample documents that have previously been confirmed in their correctness can be used to identify problems with processing software.

5. Summary and Conclusion

The CALS Initiative has gone a long way towards introducing uniform handling of text documents, in particular technical manuals, into the Armed Forces. Further progress is required, however, if the full benefits of the constituent technologies are to be realized. Further effort is required in the following areas:

1. Guidelines need to be developed to specify development procedures for creating new text markup languages.
2. Guidelines, markup specifications and publishing specifications are required for the manuals that are needed to support a text markup language.

   Like any other system of interest to the Armed Forces, a text markup language needs to be supported by technical manuals. A text markup language can be considered to be a piece of equipment in this regard. Required supporting manuals are Operation Manuals, Interfacing Manuals, Maintenance Manuals and Design Manuals.
3. Independent Verification and Validation needs to be done on all components of a text markup language, on its design and on its use.
4. The emphasis of contributions to the evolution of MIL-M-28001 has to change towards developing these guidelines and documentation specifications.

Development of these guidelines and specifications will place Armed Forces text markup language use on the firm footing required of other computer-based systems, and is the key to Tri-Service use of SGML-based text markup languages.